Habilitation to Supervise Research (HDR) Application File

Static Analyses for the Properties, Programs, and People of Tomorrow

Analyses statiques pour les propriétés, les programmes et le public de demain

Caterina Urban

April 20, 2025

Contents

Habilitation to Supervise Research (HDR) Application File

Caterina Urban

Curriculum Vitae

1

Curriculum Vitae

Personal Information

First and Last Name	**Caterina Urban**
Address	École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France
Telephone	+33 1 44 32 21 17
Email	caterina.urban@inria.fr
Webpage	https://caterinaurban.github.io
DBLP	http://dblp.org/pers/hd/u/Urban:Caterina
Google Scholar	http://scholar.google.it/citations?user=4-u1_HIAAAAJ

1.1 Professional Career

Feb 2019 - Now	**Research Scientist (Chargée de Recherche)**, Inria Paris, France Équipe ANTIQUE (DIENS, UMR 8548)
Sep 2015 - Jan 2019	**Postdoctoral Researcher**, ETH Zurich, Switzerland Mentor: Peter Müller
Mar 2015 - Jul 2015	**Research Internship**, NASA Ames Research Center, USA Mentors: Temesghen Kahsai and Arie Gurfinkel

Career Breaks

Apr 2023 - Oct 2023	**Maternity Leave**
Dec 2020 - Apr 2021	**Maternity Leave**
Sep 2018 - Dec 2018	**Maternity Leave**

1.2 Education

Dec 2011 - Jul 2015	**Ph.D. in Computer Science**, École Normale Supérieure, France Advisors: Radhia Cousot and Antoine Miné Final mark: Summa cum Laude
May 2015	Fifth Summer School on Formal Techniques, Menlo College, Atherton, USA
Fall 2009 - Fall 2011	**Master's degree in Computer Science**, University of Udine, Italy Final mark: Summa cum Laude
Fall 2006 - Fall 2009	**Bachelor's degree in Computer Science**, University of Udine, Italy Final mark: Summa cum Laude

1.3 Grants

Jan 2024 - Dec 2027	**Partner, ANR Collaborative Research Project** (PI: Aurélie Hurault, previously João Marques-Silva)
Oct 2023 - Sep 2029	**PI, Projet Ciblé du Programme de recherche Intelligence Artificielle (PEPR IA)**
Oct 2021 - Sep 2022	**Industrial Grant, Fujitsu**
2021	**Industrial Grant, Airbus**
2020	**Industrial Grant, Airbus**
Jan 2017 - Dec 2017	**PI, ETH Zurich Career Seed Grant**

1.4 Awards and Honors

Honorable Mention Award	Prix de Thèse Gilles Kahn 2015
Best Paper Award	25th Conference on Automated Deduction (CADE 2015) [c8]
Best Poster Award	37th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2011) [c1]

[c1] Miculan and Urban - Formal Analysis of Facebook Connect Single Sign-On Authentication Protocol (SOFSEM 2011)
[c8] D'Silva and Urban - Abstract Interpretation as Automated Deduction (CADE 2015)

1.5 Professional Responsibilities

Chair

General Chair

- 20th Conference on Integrated Formal Methods (iFM 2025)

Committee Chair

- Doctoral Dissertation Award @ Joint Conferences on Theory and Practice of Software 2025 (ETAPS 2025)
- Doctoral Dissertation Award @ Joint Conferences on Theory and Practice of Software 2024 (ETAPS 2024)
- Doctoral Dissertation Award @ Joint Conferences on Theory and Practice of Software 2023 (ETAPS 2023)
- Posters @ Conference on Systems, Programming, Languages, and Applications 2022 (SPLASH 2022)
- Student Research Competition @ Conference on Systems, Programming, Languages, and Applications 2022 (SPLASH 2022)
- 29th Static Analysis Symposium (SAS 2022)
- Doctoral Dissertation Award @ Joint Conferences on Theory and Practice of Software 2022 (ETAPS 2022)
- Student Research Competition @ Conference on Systems, Programming, Languages, and Applications 2021 (SPLASH 2021)
- 10th Workshop on the State Of the Art in Program Analysis (SOAP 2021)
- Doctoral Dissertation Award @ Joint Conferences on Theory and Practice of Software 2021 (ETAPS 2021)
- Doctoral Dissertation Award @ Joint Conferences on Theory and Practice of Software 2020 (ETAPS 2020)

Organizer

- Dagstuhl Seminar 25421 (`https://www.dagstuhl.de/25421`)
- Mentoring Workshop @ Joint Conferences on Theory and Practice of Software 2024 (ETAPS 2024)
- N40AI Workshop @ 51st Symposium on Principles of Programming Languages (POPL 2024)

- Mentoring Workshop @ Joint Conferences on Theory and Practice of Software 2023 (ETAPS 2023)
- Mentoring Workshop @ Federated Logic Conference 2022 (FLOC 2022)
- Mentoring Workshop @ Joint Conferences on Theory and Practice of Software 2022(ETAPS 2022)
- Mentoring Workshop @ 33rd Conference on Computer-Aided Verification (CAV 2021)

Coordinator

- Dagstuhl Seminar 16471 (https://www.dagstuhl.de/16471)

Board Member

Executive Board

- Joint Conferences on Theory and Practice of Software (2019-now)
  Responsability: Ph.D. Activities

Scientific Advisor Board

- Laboratoire Méthodes Formelles (LMF) de l'Université Paris-Saclay (2023-now)

Steering Committee Member

- Summer Schools on Foundations of Programming and Software Systems (FoPSS, 2023-now)
- Static Analysis Symposium (SAS, 2023-2028)
- Workshop on the State of the Art in Program Analysis (SOAP, 2022-2024)

Journal Editor

Associate Editor

- Transactions on Programming Languages and Systems (2023-now)

Guest Editor

- Formal Methods in System Design (Special Issue on SAS 2022)
- Formal Methods in System Design (Special Issue on CAV 2020)

Program Committee Member

Conferences

- 41st Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2026)
- 32nd Static Analysis Symposium (SAS 2025)
- 28th Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2025)
- 52nd Symposium on Principles of Programming Languages (POPL 2025)
- 25th Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2024)
- 36th Conference on Computer Aided Verification (CAV 2024)
- 30th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024)
- 35th Conference on Computer Aided Verification (CAV 2023)
- 15th NASA Formal Methods Symposium (NFM 2023)
- 32nd European Symposium on Programming (ESOP 2023)
- 19th Colloquium on Theoretical Aspects of Computing (ICTAC 2022)

- 34th Conference on Computer Aided Verification (CAV 2022)
- 49th Symposium on Principles of Programming Languages (POPL 2022)
- 25th Brazilian Symposium on Programming Languages (SBLP 2021)
- 33rd Conference on Computer Aided Verification (CAV 2021)
- 13th NASA Formal Methods Symposium (NFM 2021)
- 4th Conference on Fairness, Accountability, and Transparency 2021 (FAccT 2021)
- 27th Static Analysis Symposium (SAS 2020)
- 12th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2020)
- 32nd Conference on Computer Aided Verification (CAV 2020)
- 29th European Symposium on Programming (ESOP 2020)
- 21st Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2020)
- 15th Conference on integrated Formal Methods (iFM 2019)
- 19th Conference on Embedded Software (EMSOFT 2019)
- 29th Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)
- 31st Conference on Computer Aided Verification (CAV 2019)
- 20th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2019)
- 18th International Conference on Embedded Software (EMSOFT 2018)
- 14th Conference on integrated Formal Methods (iFM 2018)
- 25th Static Analysis Symposium (SAS 2018)
- 30th Conference on Computer Aided Verification (CAV 2018)
- 24th Static Analysis Symposium (SAS 2017)
- 23rd Static Analysis Symposium (SAS 2016)
- 17th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2016)

Workshops

- 9th Workshop on the State Of the Art in Program Analysis (SOAP 2020)
- 10th Workshop on Tools for Automatic Program Analysis (TAPAS 2019)
- 19th Workshop on Automated Verification of Critical Systems (AVoCS 2019)
- 12th Workshop on Numerical Software Verification (NSV 2019)
- 16th Workshop on Termination (WST 2018)
- 18th Workshop on Automated Verification of Critical Systems (AVoCS 2018)
- 5th Workshop on Horn Clauses for Verification and Synthesis (HCVS 2018)
- Demos @ Conference on Systems, Programming, Languages and Applications: Software for Humanity 2015 (SPLASH 2015)

Artifact Evaluations

- 19th Conference on Programming Language Design and Implementation (PLDI 2019)
- 46th Symposium on Principles of Programming Languages (POPL 2019)
- 27th Conference on Computer-Aided Verification (CAV 2015)

Competitions

- ACM Student Research Competition 2022
- Student Research Competition @ Conference on Systems, Programming, Languages, and Applications 2020 (SPLASH 2020)
- Student Research Competition @ 18th Conference on Programming Language Design and Implementation (PLDI 2018)
- 4th International Competition on Software Verification (SV-COMP 2015)

Ethical Review Committee Member

- 35th Conference on Neural Information Processing Systems (NeurIPS 2021)

Reviewer

Journals

- Foundations and Trends in Programming Languages (2025)
- Communications of the ACM (2022)
- Formal Methods in System Design (2022)
- Transactions on Programming Languages and Systems (2022)
- Software: Practice and Experience (2021)
- Transactions on Software Engineering (2018)
- Transactions on Software Engineering (2017)
- Transactions on Programming Languages and Systems (2017)
- Formal Methods in System Design (2017)
- Transactions on Programming Languages and Systems (2016)
- Acta Informatica (2016)
- Transactions on Programming Languages and Systems (2015)

Conferences

- Conference on Systems, Programming, Languages, and Applications: Software for Humanity 2022 (OOPSLA 2022)
- 22nd Conference on Formal Methods in Computer-Aided Design (FMCAD 2022)
- 28th Static Analysis Symposium (SAS 2021)
- 47th Symposium on Principles of Programming Languages (POPL 2020)
- 11th NASA Formal Methods Symposium (NFM 2019)
- 45th Symposium on Principles of Programming Languages (POPL 2018)
- 26th European Symposium on Programming (ESOP 2017)
- 18th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2017)
- 26th Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)
- 21st Conference on Formal Methods (FM 2016)
- 8th NASA Formal Methods Symposium (NFM 2016)
- 22nd Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)
- 30th Conference on Automated Software Engineering (ASE 2015)
- 22nd Static Analysis Symposium (SAS 2015)
- 27th Conference on Computer-Aided Verification (CAV 2015)
- 26th Conference on Computer-Aided Verification (CAV 2014)
- 8th Conference on Theoretical Computer Science (TCS 2014)

Research Proposals

- European Research Council (Remote Referee, 2021)

Book Manuscripts

- MIT Press (2020)

Book Chapters

- In Dirk Beyer, "Automatic Software Verification: An Overview of the State of the Art" (Springer, to appear)

Ph.D. Manuscripts

- *Pankaj Kumar Kalita*, Indian Institute of Technology Kanpur, India (2025)
- *Marco Zanella*, Università degli Studi di Padova, Italy (March 2021)

Ph.D. Jury Member

- *Linpeng Zhang*, University College London, UK (2025)
- *John Törnblom*, Linköping University, Sweden (August 2025)
- *Olivier Martinot*, Université Paris Cité, France (December 2024)
- *Guillaume Vidot*, Université Toulouse 2 - Jean Jaurès, France (December 2022)
- *Guillaume Girol*, Université Paris-Saclay, France (October 2022)
- *Julien Girard-Satabin*, Université Paris-Saclay, France (November 2021)
- *Emilio Incerto*, Gran Sasso Science Institute, Italy (April 2019)

Hiring Committee Member

- Jury d'Admissibilité CRCN/ISFP at Inria de l'Université de Lorraine (2025)
- Selection Committee for Associate Professor in Computer Science at Université de Lille (2025)
- Hiring Committee for Associate Professor in Computer Science at Université de La Réunion (2024)
- Hiring Committee for Assistant Professor in Computer Science at École Polytechnique (2024)
- Assessment Committee for Associate Professor in Systems and Software Engineering at the University of Copenhagen (2023)
- Jury d'Admissibilité CRCN/ISFP at Inria Paris (2022)

Other Committee Member

- Commission des Emplois Scientifiques at Inria Paris (2022)
- Commission des Emplois Scientifiques at Inria Paris (2021)

Panel Member

- Programming Languages Mentoring Workshop @ Conference on Systems, Programming, Languages, and Applications 2024 (SPLASH 2024)
- W@SPLASH @ Conference on Systems, Programming, Languages, and Applications 2022 (SPLASH 2022)
- VMI Career Evening @ ETH Zurich (2022)

Publicity Chair

- Federated Logic Conference 2026 (FLoC 2026)
- 25th Static Analysis Symposium (SAS 2018)
- 24th Static Analysis Symposium (SAS 2017)

1.6 Teaching Experience

Ph.D. Level

- Doctorate in Computer Science, Gran Sasso Science Institute, Italy (2021)
  Course: *Abstract Interpretation and Applications Beyond the Beaten Track* (Lectures: 6 hours)
- Doctorate in Computer Science, ETH Zurich, Switzerland (2015)
  Seminar: *Research Topics in Software Engineering*

Master Level

- Master Parisien de Recherche en Informatique (MPRI), Université de Paris, France (2024)
  Course: *Abstract Interpretation* (Lectures: 12 hours)
- Master Parisien de Recherche en Informatique (MPRI), Université de Paris, France (2023)
  Course: *Abstract Interpretation* (Lectures: 12 hours)
- Master Parisien de Recherche en Informatique (MPRI), Université de Paris, France (2022)
  Course: *Abstract Interpretation* (Lectures: 12 hours)
- Master Parisien de Recherche en Informatique (MPRI), Université de Paris, France (2021)
  Course: *Abstract Interpretation* (Lectures: 10 hours)
- Master Parisien de Recherche en Informatique (MPRI), Université de Paris, France (2020)
  Course: *Abstract Interpretation* (Lectures: 6 hours)
- Master in Computer Science, ETH Zurich, Switzerland (2017)
  Course: *Concepts of Object Oriented Programming* (Exercise Sessions: 18 hours)
- Master in Computer Science, ETH Zurich, Switzerland (2016, Head Teaching Assistant)
  Course: *Concepts of Object Oriented Programming* (Exercise Sessions: 14 hours)
- Master in Computer Science, ETH Zurich, Switzerland (2015)
  Course: *Concepts of Object Oriented Programming* (Exercise Sessions: 10.5 hours)
- Master Parisien de Recherche en Informatique (MPRI), Université de Paris, France (2014)
  Course: *Abstract Interpretation* (Lectures: 1.5 hours)

Bachelor Level

- Bachelor in Computer Science, ETH Zurich, Switzerland (2018, Head Teaching Assistant)
  Course: *Formal Methods and Functional Programming* (Exercise Sessions: 12 hours)
- Bachelor in Computer Science, ETH Zurich, Switzerland (2017)
  Course: *Formal Methods and Functional Programming* (Exercise Sessions: 12 hours)
- Bachelor in Computer Science, ETH Zurich, Switzerland (2016)
  Course: *Formal Methods and Functional Programming* (Exercise Sessions: 12 hours)
- Licence Frontières du Vivant, Université Paris Descartes (Paris 5), France (2015)
  Course: *Mathematics* (Lectures: 10 hours, Exercise Sessions: 2 hours)
- Licence Frontières du Vivant, Université Paris Descartes (Paris 5), France (2014)
  Course: *Mathematics* (Lectures: 13.5 hours, Exercise Sessions: 7 hours)
- Licence Frontières du Vivant, Université Paris Descartes (Paris 5), France (2013)
  Course: *Mathematics* (Lectures: 4 hours, Exercise Sessions: 10 hours)

Summer Schools

- 4th Summer School on Security Testing and Verification (ST&V 2025)
- Oregon Programming Languages Summer School 2025 (OPLSS 2025)
- Lipari Summer School on Abstract Interpretation 2024 (LSSAI 2024)
- Summer School on Role and effects of ARTificial Intelligence in Secure ApplicatioNs 2024 (ARTISAN 2024)
- 16th Summer School on Verification Technology, Systems & Applications (VTSA 2024)

- École Jeunes Chercheuses et Jeunes Chercheurs en Programmation 2024 (EJCP 2024)
- 4th International Programming Language Implementation Summer School (PLISS 2022)
- 13th International School of Rewriting (ISR 2022)
- 2nd Inria-DFKI European Summer School on Artificial Intelligence (IDESSAI 2022)

1.7 Mentoring Experience

Postdoc

- *Guannan Wei* (Sep 2024 - Aug 2025)
  Funding: PEPR IA
- *Alessandro De Palma* (Nov 2023 - Feb 2025)
  Funding: PEPR IA
  Publications: [j3]
- *Marco Campion* (Feb 2023 - Aug 2025)
  Funding: Fujitsu, PEPR IA
  Publications: [c22] [c24] [c25] [c28] [u1] [u2] [u3]

[j3] De Palma et al. - On Using Certified Training towards Empirical Robustness (TMLR 2025)
[c22] Campion et al. - A Formal Framework to Measure the Incompleteness of Abstract Interpretations (SAS 2023)
[c24] Campion et al. - Monotonicity and the Precision of Program Analysis (POPL 2024)
[c25] Mazzucato et al. - Quantitative Input Usage Static Analysis (NFM 2024)
[c28] Mazzucato et al. - Quantitative Static Timing Analysis (SAS 2014)
[u1] Campion et al. - Kernel Properties in Abstract Interpretation
[u2] Campion et al. - Abstract Lipschitz Continuity
[u3] Campion et al. - Measuring vs Abstracting: On the Relation between Distances and Abstract Domains

Ph.D.

- *Naïm Moussaoui Remil* (Nov 2023 - Oct, expected)
  "Static Analysis by Abstract Interpretation of Robust Temporal Properties of Programs"
  Funding: Doctoral Scholarship Inria CORDI-S
  Supervision Quota: 100%
  Publications: [c26] [u4]
- *Serge Durand* (Nov 2021 - Dec 2025, expected)
  "Formal Verification and Specification of Machine Learning Algorithms"
  Funding: CEA/DGA, PEPR IA
  Supervision Quota: 50% (co-supervision with Zakaria Chihani, CEA)
  Publications: [w3] [j3]
- *Denis Mazzucato* (Oct 2020 - Dec 2024)
  "Static Analysis by Abstract Interpretation of Quantitative Program Properties"
  Funding: Doctoral Scholarship Inria CORDI-S, PEPR IA
  Supervision Quota: 100% (HDR exemption)
  Publications: [c18] [c25] [c28]

[c26] Moussaoui Remil et al. - Automatic Detection of Vulnerable Variables for CTL Properties of Programs (LPAR 2024)
[u4] Moussaoui Remil et al. - Termination Resilience Static Analysis
[w3] Durand et al. - ReCIPH:Relational Coefficients for Input Partitioning Heuristic (WFVML 2022)
[c18] Mazzucato and Urban - Reduced Products of Abstract Domains for Fairness Certification of Neural Networks (SAS 2021)

Visiting Ph.D.

- *Giacomo Zanatta*, University of Venice, Italy (Sep 2024 - Dec 2024)
- *Greta Dolcetti*, University of Venice, Italy (Sep 2024 - Dec 2024)
- *Luca Negrini*, University of Venice, Italy (Jan 2022 - Apr 2022)
  Publications: [w4]
- *Marco Zanella*, University of Padua, Italy (May 2020 - Aug 2020)
  Publications: [c19]

[w4] Negrini et al. - Static Analysis of Data Transformations in Jupyter Notebooks (SOAP 2023)
[c19] Ranzato et al. - Fairness-Aware Training of Decision Trees by Abstract Interpretation (CIKM 2021)

Master

- *Thomas Winninger*, Télécom SudParis, France (Mar 2025 - May 2025)
  M2 Research Internship
  Supervision Quota: 100%
- *Pierre Goutagny*, École Normale Supérieure de Lyon, France (Mar 2024 - Jul 2024)
  M2 Research Internship
  Supervision Quota: 100%
- *Loïc Chevalier*, École Normale Supérieure, France (Spring 2024)
  Supervised Research Project
  Supervision Quota: 100%
- *Naïm Moussaoui-Remil*, École Normale Supérieure de Rennes, France (Mar 2023 - Aug 2023)
  M2 Research Internship
  Supervision Quota: 50% (co-supervised with Antoine Miné, Sorbonne University)
  Publications: [c26]
- *Kevin Pinochet*, University of Chile, Chile (Jan 2023 - Apr 2023)
  Research Internship
  Supervision Quota: 100%
- *Ali El Husseini*, École Normale Supérieure Paris-Saclay, France (Mar 2022 - Aug 2022)
  M2 Research Internship
  Supervision Quota: 100%
- *Serge Durand*, École Normale Supérieure Paris-Saclay, France (Jun 2020 - Aug 2020)
  M1 Research Internship
  Supervision Quota: 100%
  Publications: [w3]
- *Lowis Engel*, ETH Zurich, Switzerland (Feb 2018 - Aug 2018)
  Master's Thesis
  Supervision Quota: 100%
- *Madelin Schumacher*, ETH Zurich, Switzerland (Sep 2017 - Mar 2018)
  Master's Thesis'
  Supervision Quota: 50% (co-supervised with Alexandra Bugariu, ETH Zurich)
- *Samuel Ueltschi*, ETH Zurich, Switzerland (Mar 2017 - Sep 2017)
  Master's Thesis
  Supervision Quota: 100%
  Publications: [c15]
- *Simon Wehrli*, ETH Zurich, Switzerland (Feb 2017 - Aug 2017)
  Master's Thesis
  Supervision Quota: 100%
- *Flurin Rindisbacher*, ETH Zurich, Switzerland (Mar 2017 - Aug 2017)
  Master's Thesis
  Supervision Quota: 100% (co-supervised with Jérôme Dohrau, ETH Zurich)
- *Severin Münger*, ETH Zurich, Switzerland (Sep 2016 - Mar 2017)
  Master's Thesis
  Supervision Quota: 50% (co-supervised with Alexander J. Summers, ETH Zurich)
  Publications: [c13]
- *Lukas Neukom*, ETH Zurich, Switzerland (Mar 2016 - Sep 2016)
  Master's Thesis
  Supervision Quota: 100%
- *Seraiah Walter*, ETH Zurich, Switzerland (Feb 2016 - Aug 2016)
  Master's Thesis
  Supervision Quota: 50% (co-supervised with Alexander J. Summers, ETH Zurich)

[c26] Moussaoui Remil et al. - Automatic Detection of Vulnerable Variables for CTL Properties of Programs (LPAR 2024)
[w3] Durand et al. - ReCIPH:Relational Coefficients for Input Partitioning Heuristic (WFVML 2022)
[c15] Urban et al. - Abstract Interpretation of CTL Properties (SAS 2018).
[c13] Dohrau et al. - Permission Inference for Array Programs (CAV 2018)

Bachelor

- *Abhinandan Pal*, IIIT Kalyani, India (Nov 2022 - Jan 2023)
  Research Internship
  Supervision Quota: 100%
- *Abhinandan Pal*, IIIT Kalyani, India (May 2022 - Jul 2022)
  Research Internship
  Supervision Quota: 100%
- *Guruprerana Shabadi*, École Polytechnique, France (Jan 2022 - Mar 2022)
  L3 Research Internship
  Supervision Quota: 100%
- *Abhinandan Pal*, IIIT Kalyani, India (Dec 2021 - Jan 2022)
  Research Internship
  Supervision Quota: 100%
  Publications: [c23]
- *Radwa Sherif Abdelbar*, German University in Cairo, Egypt (Mar 2018 - Aug 2018)
  Bachelor's Thesis
  Supervision Quota: 50% (co-supervised with Alexandra Bugariu, ETH Zurich)
- *Mostafa Hassan*, German University in Cairo, Egypt (Mar 2017 - Aug 2017)
  Bachelor's Thesis
  Supervision Quota: 50% (co-supervised with Marco Eilers, ETH Zurich)
  Publications: [c14]
- *Nathanaëlle Courant*, École Normale Supérieure, France (Jun 2016 - Jul 2016)
  L3 Research Internship
  Supervision Quota: 100%
  Publications: [c11]

[c23] Pal et al. - Abstract Interpretation-Based Feature Importance for Support Vector Machines (VMCAI 2024)
[c14] Hassan et al. - MaxSMT-Based Type Inference for Python 3 (CAV 2018)
[c11] Courant and Urban - Precise Widening Operators for Proving Termination by Abstract Interpretation (TACAS 2017)

1.8 Publications

International Journals

[j3] Alessandro De Palma, Serge Durand, Zakaria Chihani, François Terrier, **Caterina Urban**. On Using Certified Training towards Empirical Robustness.
In Transactions on Machine Learning Research (TMLR), 2025.
https://openreview.net/pdf?id=UaaT2fI9DC

[j2] Vijay D'Silva, **Caterina Urban**. Abstract Interpretation as Automated Deduction.
In Journal of Automated Reasoning (JAR), 2017.
https://caterinaurban.github.io/publication/jar2017/

[j1] **Caterina Urban**, Antoine Miné, Inference of Ranking Functions for Proving Temporal Properties by Abstract Interpretation.
In Computer Languages Systems and Structures (COMLAN), 2017.
https://inria.hal.science/hal-01312239/

International Conferences

[c28] Denis Mazzucato, Marco Campion, **Caterina Urban**. Quantitative Static Timing Analysis.
In 31st Static Analysis Symposium (SAS 2024).
https://inria.hal.science/hal-04669723
Radhia Cousot Best Paper Award for Denis Mazzucato
Awarded the Validated, Extensible, and Available Artifact Evaluation Badges

[c27] Filip Drobnjakovic, Pavle Subotic, **Caterina Urban**. An Abstract Interpretation-Based Data Leakage Static Analysis.
In 18th Symposium on Theoretical Aspects of Software Engineering (TASE 2024).
https://inria.hal.science/hal-04556578

[c26] Naïm Moussaoui Remil, **Caterina Urban**, Antoine Miné. Automatic Detection of Vulnerable Variables for CTL Properties of Programs.
In 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2024).
https://inria.hal.science/hal-04710215

[c25] Denis Mazzucato, Marco Campion, **Caterina Urban**. Quantitative Input Usage Static Analysis.
In 16th NASA Formal Methods Symposium (NFM 2024).
https://hal.science/hal-04339001

[c24] Marco Campion, Mila Dalla Preda, Roberto Giacobazzi, **Caterina Urban**. Monotonicity and the Precision of Program Analysis.
In 51st Symposium on Principles of Programming Languages (POPL 2024).
https://inria.hal.science/hal-04423578

[c23] Abhinandan Pal, Francesco Ranzato, **Caterina Urban**, Marco Zanella. Abstract Interpretation-Based Feature Importance for Support Vector Machines.
In 25th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2024).
https://inria.hal.science/hal-04378817

[c22] Marco Campion, **Caterina Urban**, Mila Dalla Preda, Roberto Giacobazzi. A Formal Framework to Measure the Incompleteness of Abstract Interpretations.
In 30th Static Analysis Symposium (SAS 2023).
https://inria.hal.science/hal-04249990

[c21] Satoshi Munakata, **Caterina Urban**, Haruki Yokoyama, Koji Yamamoto, Kazuki Munakata. Verifying Attention Robustness of Deep Neural Networks against Semantic Perturbations.
In 15th NASA Formal Methods Symposium (NFM 2023)
https://inria.hal.science/hal-04249934

[c20] Satoshi Munakata, **Caterina Urban**, Haruki Yokoyama, Koji Yamamoto, Kazuki Munakata. Verifying Attention Robustness of Deep Neural Networks against Semantic Perturbations (Poster).
In 29th Asia-Pacific Software Engineering Conference (APSEC 2022).

[c19] Francesco Ranzato, **Caterina Urban**, Marco Zanella. Fairness-Aware Training of Decision Trees by Abstract Interpretation.
In 30th International Conference on Information and Knowledge Management (CIKM 2021).
https://inria.hal.science/hal-03545701

[c18] Denis Mazzucato, **Caterina Urban**. Reduced Products of Abstract Domains for Fairness Certification of Neural Networks.
In 28th Static Analysis Symposium (SAS 2021).
https://inria.hal.science/hal-03348036
Awarded the Validated, Extensible, and Available Artifact Evaluation Badges

[c17] **Caterina Urban**, Maria Christakis, Valentin Wüstholz, Fuyuan Zhang. Perfectly Parallel Fairness Certification of Neural Networks.
In ACM on Programming Languages (PACMPL), Conference on Object-Oriented Programming Systems, Languages, and Applications 2020 (OOPSLA 2020).
https://inria.hal.science/hal-03091870
Awarded the Functional, Reusable, and Available Artifact Evaluation Badges

[c16] **Caterina Urban**. Static Analysis of Data Science Software (Invited Paper).
In 26th Static Analysis Symposium (SAS 2019).
https://inria.hal.science/hal-02397699

[c15] **Caterina Urban**, Samuel Ueltschi, Peter Müller. Abstract Interpretation of CTL Properties
In 25th Static Analysis Symposium (SAS 2018).
https://caterinaurban.github.io/publication/sas2018/
Awarded the Artifact Evaluation Badge

[c14] Mostafa Hassan, **Caterina Urban**, Marco Eilers, Peter Müller. MaxSMT-Based Type Inference for Python

3
In 30th Conference on Computer Aided Verification (CAV 2018).
https://caterinaurban.github.io/publication/cav2018a/
Awarded the Artifact Evaluation Badge

[c13] Jérôme Dohrau, Alexander J. Summers, **Caterina Urban**, Severin Münger, Peter Müller. Permission Inference for Array Programs.
In 30th Conference on Computer Aided Verification (CAV 2018).
https://caterinaurban.github.io/publication/cav2018b/
Awarded the Artifact Evaluation Badge
Invited Talk at 1st Workshop on Parallel Logical Reasoning (PLR 2018)

[c12] **Caterina Urban**, Peter Müller. An Abstract Interpretation Framework for Input Data Usage.
In 27th European Symposium on Programming (ESOP 2018).
https://caterinaurban.github.io/publication/esop2018/

[c11] Nathanaëlle Courant, **Caterina Urban**. Precise Widening Operators for Proving Termination by Abstract Interpretation.
In 23rd Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017).
https://caterinaurban.github.io/publication/tacas2017/

[c10] Vijay D'Silva, **Caterina Urban**. Büchi, Lindenbaum, Tarski: A Program Analysis Appetizer (Invited Paper).
In 25th Joint Conference on Artificial Intelligence (IJCAI 2016).
https://caterinaurban.github.io/publication/ijcai2016/

[c9] **Caterina Urban**, Arie Gurfinkel, Temesghen Kahsai. Synthesizing Ranking Functions from Bits and Pieces.
In 22nd Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016).
https://caterinaurban.github.io/publication/tacas2016/

[c8] Vijay D'Silva, **Caterina Urban**. Abstract Interpretation as Automated Deduction.
In 25th Conference on Automated Deduction (CADE 2015).
https://inria.hal.science/hal-01952896
Best Paper Award

[c7] Vijay D'Silva, **Caterina Urban**. Conflict-Driven Abstract Interpretation for Conditional Termination.
In 27th Conference on Computer Aided Verification (CAV 2015).
https://inria.hal.science/hal-01952911

[c6] **Caterina Urban**. FuncTion: An Abstract Domain Functor for Termination
In 21st Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015).
https://inria.hal.science/hal-01107419

[c5] **Caterina Urban**, Antoine Miné. Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation
In 16th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2015).
https://inria.hal.science/hal-01105238

[c4] **Caterina Urban**, Antoine Miné. A Decision Tree Abstract Domain for Proving Conditional Termination.
In 21st Static Analysis Symposium (SAS 2014).
https://inria.hal.science/hal-01105221

[c3] **Caterina Urban**, Antoine Miné. An Abstract Domain to Infer Ordinal-Valued Ranking Functions
In 23rd European Symposium on Programming (ESOP 2014).
https://inria.hal.science/hal-00925731

[c2] **Caterina Urban**. The Abstract Domain of Segmented Ranking Functions
In 20th Static Analysis Symposium (SAS 2013).
https://inria.hal.science/hal-00925670

[c1] Marino Miculan, **Caterina Urban**. Formal Analysis of Facebook Connect Single Sign-On Authentication Protocol.

In Student Research Forum of 37th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2011).
https://caterinaurban.github.io/publication/sofsem2011/
Best Poster Award

International Workshops

[w5] Greta Dolcetti, Agostino Cortesi, **Caterina Urban**, Enea Zaffanella. Towards a High Level Linter for Data Science.
In 10th Workshop on Numerical and Symbolic Abstract Domains (NSAD 2024).
https://inria.hal.science/hal-04739441

[w4] Luca Negrini, Guruprerana Shabadi, **Caterina Urban**. Static Analysis of Data Transformations in Jupyter Notebooks.
In 12th Workshop on the State Of the Art in Program Analysis (SOAP 2023).
https://inria.hal.science/hal-04249950

[w3] Serge Durand, Augustin Lemesle, Zakaria Chihani, **Caterina Urban**, François Terrier. ReCIPH: Relational Coefficients for Input Partitioning Heuristic (Poster).
In 1st Workshop on Formal Verification of Machine Learning (WFVML 2022).
https://inria.hal.science/hal-03926281

[w2] **Caterina Urban**, Antoine Miné, To Infinity. . . and Beyond!
In 14th Workshop on Termination (WST 2014).
https://inria.hal.science/hal-01105216

[w1] **Caterina Urban**. Piecewise-Defined Ranking Functions
In 13th Workshop on Termination (WST 2013).
https://inria.hal.science/hal-00925682

Book Chapters

[b2] **Caterina Urban**. The FuncTion Static Analyzer.
In Dirk Beyer, "Automatic Software Verification: An Overview of the State of the Art'. To appear.

[b1] **Caterina Urban**. Static Analysis for Data Scientists.
In Vincenzo Arceri, Agostino Cortesi, Pietro Ferrara, Martina Olliaro, "Challenges of Software Verification", Intelligent Systems Reference Library, Volume 238, 2023.
https://inria.hal.science/hal-04249957

Other Publications

[o2] **Caterina Urban**. Analyse Statique par Interprétation Abstraite de Propriétés Temporelles des Programmes.
In 1024 - Bulletin de la Société Informatique de France, April 2016.
https://1024.socinfo.fr/2016/04/1024_8_2016_133.pdf

[o1] **Caterina Urban**. Ce Qu'Achille a Fait Calculer à la Tortue.
In Blog Binaire, March 2016.
http://binaire.blog.lemonde.fr/2016/03/25/ce-quachille-a-fait-calculer-a-la-tortue/

Under Submission

[u4] Naïm Moussaoui Remil, **Caterina Urban**. Termination Resilience Static Analysis.

[u3] Marco Campion, Isabella Mastroeni, **Caterina Urban**. Measuring vs Abstracting: On the Relation between Distances and Abstract Domains.

[u2] Marco Campion, Isabella Mastroeni, Michele Pasqua, **Caterina Urban**. Abstract Lipschitz Continuity
https://inria.hal.science/hal-04935306

[u1] Marco Campion, Mila Dalla Preda, Roberto Giacobazzi, **Caterina Urban**. Kernel Properties in Abstract Interpretation.

Habilitation to Supervise Research (HDR) Application File

Caterina Urban

Summary Report

Mémoire de synthèse

2

Career and Work

Parcours et travaux

I am interested in developing methods and tools to enhance the quality and reliability of computer software and, more broadly, to help understanding complex software systems. My main area of expertise is *static analysis*, particularly within the framework of *abstract interpretation*, a unifying mathematical theory for describing and comparing the behavior of computer programs at different levels of abstraction. However, in my research activity, I have also gained expertise in *software model checking* and *deductive program verification*. I thus have a broad knowledge that spans the whole spectrum of formal methods.

Software defects can have catastrophic consequences in safety-critical areas such as transportation, nuclear power generation, and medical systems. Moreover, as our reliance on software systems grows, we become increasingly vulnerable to software defects in our daily lives. This risk is further amplified nowadays by the rise of machine learning-based software, which plays an increasingly important role in high-stakes decision making, despite its often opaque and unpredictable behavior. Ensuring the robustness, reliability, and explainability of such systems is an urgent challenge, and has been a central focus of my research since 2018.

2.1 Pushing for More Advanced Static Analyses

Proving that computer programs behave correctly is a difficult problem because it is mathematically *undecidable*: it is impossible to construct an algorithm that always gives a precise yes-or-no answer regarding the correctness of a program. Static program analysis offers a solution to this problem by relaxing the requirement of returning a yes-or-no answer: the reasoning is based on an automatically computed approximation of the behavior of the given program, which results in a positive *yes* answer (if all computed executions satisfy the desired correctness property, cf. Figure 2.1) or an inconclusive *unknown* answer (otherwise, cf. Figure 2.2). In the latter case, the analysis returns a warning, which either corresponds to a real error in the program or is a false positive due to the approximation introduced by the analysis (e.g., the red region in Figure 2.2). The main challenge is then to develop static analyses that are precise enough to return an inconclusive answer as seldom as possible.

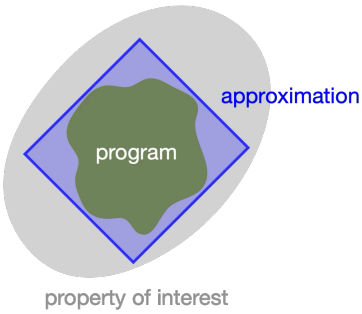


Figure 2.1: Sound approximation.

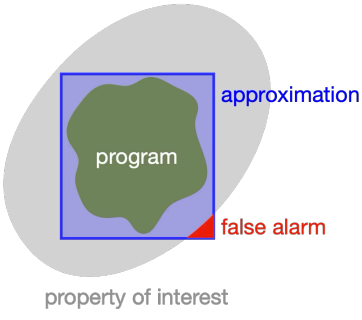
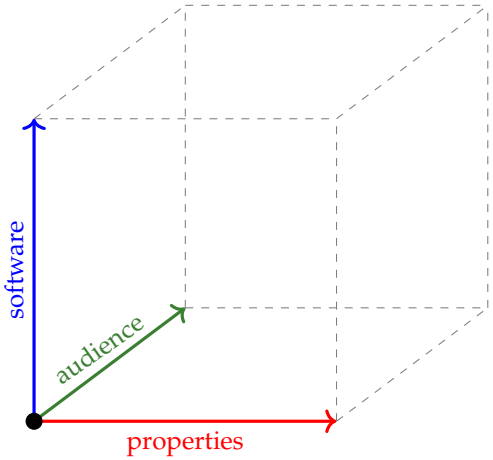


Figure 2.2: Incomplete approximation.

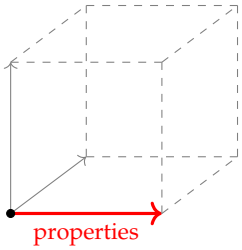
The overwhelming majority of the research in static analysis in the literature has focused on *safety correctness properties*, e.g., absence of run-time errors (such as divisions by zero, out-of-bound memory accesses, or assertion violations). This focus is well-justified for safety-critical software, where correctness violations can lead to catastrophic failures. However, modern software systems increasingly demand guarantees

that go beyond safety, and the demand for these guarantees has been democratized beyond traditional safety-critical domains. There is a growing need for static analyses that can prove more advanced *properties* of programs, as well as for static analyses that are more general-purpose, that is, applicable to a variety of *software* programs, and accessible to a broad *audience* rather than limited to experts.



Over the course of my research career, I have made contributions in all of these three directions. Most of my publications comprise solid theoretical contributions and rigorous practical implementation efforts.

Liveness Properties. With my doctoral dissertation [phd], I broadened the scope of static analysis by abstract interpretation to *termination* [c2] and other *liveness properties* [c5] of programs. The main insight that made this work possible was to relax some of the requirements imposed to the extrapolation technique (called *widening*) traditionally used to enforce convergence of the analysis, and allow the analysis to temporarily explore incorrect approximations of the behavior of a program [c11]. I implemented this work in the tool FuncTion [c6]. Additionally, I was able to transfer the main ideas at the root of this work into the context of software model checking, where I designed an approach for proving both termination and non-termination and implemented it in SeaHorn [c9].



[phd] Urban - Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs (École Normale Supérieure, 2015)
[c2] Urban - The Abstract Domain of Segmented Ranking Functions (SAS 2013)
[c5] Urban and Miné - Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation (VMCAI 2015)
[c6] Urban - FuncTion: An Abstract Domain Functor for Termination (TACAS 2015)
[c9] Urban at al. - Synthesizing Ranking Functions from Bits and Pieces (TACAS 2016).
[c11] Courant and Urban - Precise Widening Operators for Proving Termination by Abstract Interpretation (TACAS 2017)

Functional Properties. Building upon my doctoral work, I proposed a general static analysis framework for verifying *functional properties* of programs expressed in *computation tree logic* (CTL) [c15]. This generalization required rethinking the fixpoint computation strategy to account for the path quantifiers and temporal operators intrinsic to CTL (cf. Section 5.1).

Up to and including this work, I had only considered demonic non-determinism, i.e., assuming that non-deterministic program variables are controlled by an external adversary (e.g., attacker, scheduler, etc.). More recently, I became interested in studying different flavors of non-determinism, thus also considering uncontrolled non-deterministic variables (e.g., random seeds, etc.). With one of my Ph.D. student (Naïm Moussaoui Remil), I proposed a static analysis to automatically infer the minimal set of program variables that need to be controlled to ensure a program property expressed in CTL [c26]. This work opened up interesting connections to security, i.e., identifying attack vectors that

[c15] Urban et al. - Abstract Interpretation of CTL Properties (SAS 2018)
[c26] Moussaoui Remil et al. - Automatic Detection of Vulnerable Variables for CTL Properties of Programs (LPAR 2024)

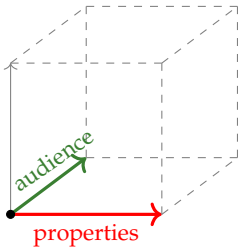
an adversary could exploit, and explainability, i.e., identifying which variables are critical to the manifestation of a particular program behavior, that I am currently exploring (cf. Section 5.2).

Hyperproperties. The shift to considering different kinds of non-deterministic program variables has driven this line of my research beyond safety and liveness functional properties, bringing me to currently investigate a broader range of program properties. These are *properties of sets of executions*, often called *hyperproperties*, including *termination resilience* – the impossibility for an adversary to cause definite non-termination of a program [u4] – and *functional non-exploitability* – the impossibility for an adversary to cause a program to violate or satisfy a functional property.

[c24] Campion et al. - Monotonicity and the Precision of Program Analysis (POPL 2024)
[u1] Campion et al. - Kernel Properties in Abstract Interpretation
[u4] Moussaoui Remil and Urban - Termination Resilience Static Analysis

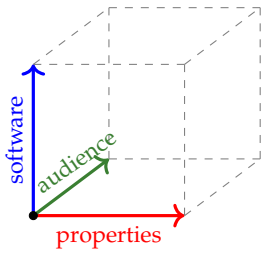
With one of my postdocs (Marco Campion) and other collaborators, I studied the impact of extensional program properties (i.e., relative to its observable input-output behavior) on the precision of a static analysis of the program, another important hyperproperty (called *completeness*), fundamental for understanding the limits of a static program analysis. In particular, we showed that a static analysis of programs (or program fragments) that manifest a monotone behavior will not produce false alarms [c24]. We further generalized this work showing that sufficient conditions for completeness arise from extensional program properties over a restricted subset of representative program inputs [u1].

Data Science Software. Several other compelling hyperproperties emerge in the context of data science software, notably software programs used to gather, triage, and prepare data within machine learning development pipelines. These programs – most often Python or R scripts – are written by domain experts rather than software engineers, making them especially prone to subtle errors. Supported by a Career Seed Grant awarded to me by ETH Zurich, I initiated the study of hyperproperties related to *data usage*, i.e., *how* input data is used by a program. In particular, I proposed an abstract interpretation framework for reasoning about dependencies between program variables, with the goal of detecting input data that remains unused by a program [c12]. This framework provides a unifying approach to harness various existing dependency-based analyses for data usage, such as non-interference analyses in security, strongly live variable analysis in compilers, and dependency analyses used in backward program slicing (cf. Section 5.3).



[c12] Urban and Müller - An Abstract Interpretation Framework for Input Data Usage (ESOP 2018)
[c27] Drobnjakovic et al. - An Abstract Interpretation-Based Data Leakage Static Analysis (TASE 2024)

More recently, together with collaborators at Microsoft Research, I generalized this framework to reason about dependencies between multi-dimensional program variables. I thus proposed an abstract interpretation-based static analysis to prove the *absence of data leakage*, which is an instance of data (non-)usage ensuring that datasets used for training and testing machine learning models remain independent [c27]. Data leakage can severely compromise the generalization capabilities of machine learning models, resulting in overly optimistic performance estimates during model evaluation, and leading to flawed decision-making with dangerous consequences when models are deployed in high-stakes applications.



Machine Learning Software. I studied another notable instance of data (non-)usage in the context of machine learning, notably applied to neural networks, to certify them to be free from algorithmic bias. Specifically, I

designed and developed a static analysis to prove that neural network predictions are independent from the values of certain sensitive input features [c17]. This property, called *dependency fairness*, is challenging to verify because it is a form of global robustness across the input space. What enabled a practical static analysis solution was a clever combination of forward and backward analyses to soundly parallelize and at the same time reduce the overall analysis effort (cf. Section 5.4). This work found a real-world application through a collaboration with Airbus, where we applied the analysis to certify dependency fairness of a neural network surrogate for aircraft braking distance estimation.
Through another industrial collaboration with Fujitsu, I was confronted with the inadequacy of ensuring robustness of the model prediction – a predominant focus in the literature in formal methods for machine learning – to provide guarantees of trustworthiness in practical applications. In response, we shifted the focus of the verification problem from the robustness of the model prediction to the *robustness of the explanation* of the model prediction [c21]. We proposed a first verification approach, while further more scalable approaches are ongoing work.

This work was pivotal in steering my research interests towards the application of formal methods to aid *machine learning explainability* [c23]. It has since become a major line of research in the SAIF project, which I lead within the national Priority Research Programme and Equipments on Artificial Intelligence (PEPR IA), funded by the France 2030 investment plan. Certified implementations of formal explainability techniques using the Rocq proof assistant are the object of a collaboration with the IRIT Computer Science Research Institute of Toulouse as part of the ForML project, funded by the French National Research Agency.

Motivated by the persistent divide between the formal methods and machine learning research communities, I have also been actively exploring how static analysis techniques can be leveraged *during the model training process* to achieve practical objectives of interest to the machine learning community, such as reducing the size of models [c19], or improving empirical robustness against adversarial attacks [j3].

[j3] De Palma et al. - On Using Certified-Training towards Empirical Robustness (TMLR 2025)
[c17] Urban et al. - Perfectly Parallel Fairness Certification of Neural Networks (OOPSLA 2020)
[c19] Ranzato et al. - Fairness-Aware Training of Decision Trees by Abstract Interpretation (CIKM 2021)
[c21] Munakata et al. - Verifying Attention Robustness of Deep Neural Networks against Semantic Perturbations (NFM 2023)
[c23] Pal et al. - Abstract Interpretation-Based Feature Importance for Support Vector Machines (VMCAI 2024)

Quantitative Properties. In several practical contexts, qualitative program properties – binary in nature, either satisfied or violated – are too strong. It is often acceptable or even expected to ask *how far* a property is from being satisfied or violated. For instance, it is well-known that completeness in static program analysis is extremely hard or even impossible to achieve in non-trivial cases. With one of my postdocs, we proposed a general framework for measuring the imprecision of static program analyses using pre-metrics [c22]. This quantitative approach enables a graded interpretation of the precision of a static analysis, called *partial completeness*: the static analysis is incomplete in general but its imprecision is sufficiently small to be acceptable for practical purposes. The use of pre-metrics in this work has inspired an ongoing line of research with other collaborators, studying novel interesting hyperproperties that arise from combining qualitative and quantitative approximations, such as *abstract Lipschitz continuity* [u2] and *partial abstract non-interference* [u3].

In parallel, with a recently graduated Ph.D. student (Denis Mazzucato), I proposed a static analysis framework to determine *to what degree* a program input contributes to its behavior [c25] – a quantitative form of

[c22] Campion et al. - A Formal Framework to Measure the Incompleteness of Abstract Interpretations (SAS 2023)
[c25] Mazzucato et al. - Quantitative Input Usage Static Analysis (NFM 2024)
[u2] Campion et al. - Abstract Lipschitz Continuity
[u3] Campion et al. - Measuring vs Abstracting: On the Relation between Distances and Abstract Domains

data usage. One instance of this framework, focusing on the impact of program inputs on the global number of loop iterations of the program, enabled us to formally prove that the real-world cryptographic library S2N-BIGNUM is immune to timing side-channel attacks [c28].

[c28] Mazzucato et al. - Quantitative Static Timing Analysis (SAS 2024)

2.2 Combining Static Analysis and Other Formal Methods

While the core of my research focuses on static analysis, I have also explored other formal methods and, in particular, their relation and the possible means of combining them with static analysis.

Software Model Checking. In parallel to my doctoral studies, with a collaborator, we used classic results in logic to establish precise correspondences between the mathematical foundations of abstract interpretation-based static analysis and model checking, with the objective of promoting the exchange of techniques between them and combining their complementary strengths [c8]. First, building on a classical result of Büchi that relates automata and logic, we encoded reachability as the satisfiability of formulas in a weak monadic second-order logic. Thus, we formally characterized static analyzers as solvers for the satisfiability of this family of formulas. Second, we gave a logical characterization of certain lattices used in static analyzers by using a construction of Lindenbaum and Tarski for generating lattices from logics. We thus showed that these lattices are subclassical fragments of first-order theories. This research, giving a logical characterization of abstract interpretation, led to practical applications in termination analysis: by integrating conflict-driven learning procedures – integral to the performance of SAT and SMT solvers used for model checking – we were able to drastically improve the precision of the FUNCTION termination static analyzer [c7].

[c7] D'Silva and Urban - Conflict-Driven Abstract Interpretation for Conditional Termination (CAV 2015)
[c8] D'Silva and Urban - Abstract Interpretation as Automated Deduction (CADE 2015)

Deductive Program Verification. Finally, during my postdoc at ETH Zurich, I worked on using static analysis to automatically infer specifications of (certain aspects of) the behavior of a program directly from its source code in order to reduce the effort required to formally verify that a program complies with its intended behavior. In particular, with other colleagues, we proposed a static analysis to automatically infer the memory footprint of an array-manipulating program, and produce specifications in the form of read and write access permissions that can be directly leveraged by deductive program verifiers [c13].

[c13] Dohrau et al. - Permission Inference for Array Programs (CAV 2018)

3

Collective Responsibilities

Responsabilités collectives

Throughout my academic career, I have devoted considerable effort to the collective functioning of the scientific community. These responsibilities, which complement my research and teaching activities, contribute to the organization and animation of our discipline. They also reflect my commitment to supporting others and helping the community grow.

3.1 Selection and Evaluation Committees

I have been a member of several *hiring and selection committees* for academic positions in computer science. These include admissibility juries for research scientist positions (CRCN/ISFP) at Inria, and selection committees for assistant and associate professor positions at diverse institutions, both in France and abroad. I also served as a member of the Commission des Emplois Scientifiques at Inria Paris in 2021 and 2022.

In addition, I have contributed to the evaluation of doctoral candidates through service on several *Ph.D. defense committees*, in France and internationally. In some cases, I acted as an official *reviewer of Ph.D. manuscripts*.

These responsibilities testify to the trust placed in my judgment for evaluating candidates at critical stages of their academic careers.

Hiring and Selection Committees

- Inria de l'Université de Lorraine (2025)
- Université de Lille (2025)
- Université de La Réunion (2024)
- École Polytechnique (2024)
- University of Copenhagen (2023)
- Inria Paris (2022)

Ph.D. Jury Member

- Linpeng Zhang (UK, 2025)
- John Törnblom (Sweden, 2025)
- Olivier Martinot (France, 2024)
- Guillaume Vidot (France, 2022)
- Guillaume Girol (France, 2022)
- Julien Girard-Satabin (France, 2021)
- Emilio Incerto (Italy, April 2019)

Ph.D. Manuscript Reviewer

- Pankaj Kumar Kalita (India, 2025)
- Marco Zanella (Italy, 2021)

3.2 Organization of Scientific Events

I have been involved in the organization of a number of scientific conferences, workshops, and invitational seminars. I am serving as *general chair* of the 20th International Conference on Integrated Formal Methods (iFM 2025), and I served as co-*chair of the program committee* of the 29th Static Analysis Symposium (SAS 2022) and the 10th Workshop on the State of the Art in Program Analysis (SOAP 2021). I have thus been a *member of the steering committee* of SOAP (from 2022 to 2024) and SAS (from 2023 until 2028). I have contributed as organizer of the posters and student research competition tracks at SPLASH 2022 and SPLASH 2021-2022, respectively. I have also organized scientific seminars (Dagstuhl Seminar 25421) and thematic workshops (N40AI at POPL 2024). I acted as coordinator for Dagstuhl Seminar 16471.

I also serve as a *member of the executive board* of ETAPS since 2019, where I am responsible for the Ph.D. activities of ETAPS. In particular, I started and I chair the annual ETAPS doctoral dissertation award since 2020. In addition, since 2023, I am the ETAPS representative on the

steering committee of the series of Summer Schools on Foundations of Programming and Software Systems (FoPSS).

3.3 Peer Review and Editorial Service

My involvement in *program committees* spans over 40 instances, across conferences – including flagship conferences such as POPL and CAV – as well as workshops, artifact evaluations, and competitions.

I frequently serve as a *reviewer* for international journals and conferences in programming languages, formal methods, and software engineering. I have reviewed book manuscripts (for MIT Press) and book chapters (for Springer) and acted as a remote referee for ERC proposals.

In addition to reviewing, I serve as *associate editor* for the journal Transactions on Programming Languages and Systems (TOPLAS) since 2023, and I served as *guest editor* for two special issues of the journal Formal Methods in System Design (FMSD) [Albarghouthi24, Pichardie25].

3.4 Mentoring Initiatives

I consider mentoring an essential and rewarding part of academic life. It is not only a way to support individual growth of early-career researchers, but also a key responsibility in fostering a thriving, inclusive, and supportive research community.

I co-organized several *mentoring workshops* at ETAPS, CAV, and FLoC. I have also been invited as a *panelist* at career development and networking events at SPLASH and ETH Zurich. I participate as a *mentor* in the SIGPLAN-M long-term mentoring program.

Each student at the École Normale Supérieure is supported by a *tutor* who provides guidance throughout their studies and helps them shape their academic projects. Since joining Inria in 2019 and becoming a permanent member of ENS, I have actively taken part in tutoring students.

3.5 Other Service

I participated in the *ethical review committee* of the 35th Conference on Neural Information Processing Systems (NeurIPS 2021).

I have taken up *publicity chair* roles for the 24th Static Analysis Symposium (SAS 2017) and 25th Static Analysis Symposium (SAS 2018), as well as for the 9th Federated Logic Conference (FLoC 2026).

Since 2023, I am a *member of the scientific advisory board* of the Formal Methods Laboratory (LMF) of the Université Paris-Saclay.

Program Committee Member

- **Conferences:** OOPSLA 2026, SAS 2025, FoSSaCS 2025, POPL 2025, LPAR 2024, CAV 2024, TACAS 2024, CAV 2023, NFM 2023, ESOP 2023, ICTAC 2022, CAV 2022, POPL 2022, SBLP 2021, CAV 2021, NFM 2021, FAccT 2021, SAS 2020, VSTTE 2020, CAV 2020, ESOP 2020, VMCAI 2020, iFM 2019, EMSOFT 2019, LOPSTR 2019, CAV 2019, VMCAI 2019, EMSOFT 2018, iFM 2018, SAS 2018, CAV 2018, SAS 2017, SAS 2016, VMCAI 2016
- **Workshops:** SOAP 2020, TAPAS 2019, AVoCS 2019, NSV 2019, WST 2018, AVoCS 2018, HCVS 2018, SPLASH 2015 Demos
- **Artifact Evaluations:** PLDI 2019 Artifact Evaluation, POPL 2019 Artifact Evaluation, CAV 2015 Artifact Evaluation
- **Competitions:** ACM SRC 2022, SPLASH 2020 SRC, PLDI 2018 SRC, SV-COMP 2015

Reviewer

- **Journals**: FnTs (2025), CACM (2022), FMSD (2022), TOPLAS (2022), SPE (2021), TSE (2018), TSE (2018), TOPLAS (2017), FMSD (2017), TOPLAS (2016), Acta Informatica (2016), TOPLAS (2015)
- **Conferences:** OOPSLA 2022, FMCAD 2022, SAS 2021, POPL 2020, NFM 2019, POPL 2018, ESOP 2017, VMCAI 2017, LOPSTR 2016, FM 2016, NFM 2016, TACAS 2016, ASE 2015, SAS 2015, CAV 2015, CAV 2014, TCS 2014

[Albarghouthi24] Albarghouthi et al. - Preface of the special issue on the conference on Computer-Aided Verification 2020 and 2021 (FMSD, 2024)
[Pichardie25] Pichardie et al. - Preface of the special issue on the static analysis symposium 2020 and 2022 (FMSD, 2025)

Mentoring Workshop Co-Organizer

- ETAPS Mentoring Workshop 2024
- ETAPS Mentoring Workshop 2023
- FLoC Mentoring Workshop 2022
- ETAPS Mentoring Workshop 2022
- CAV Mentoring Workshop 2021

Panel Member

- PLMW @ SPLASH 2024
- W @ SPLASH 2022
- VMI Career Event @ ETH Zurich

4

Invited Talks

Conférences invitées

Unless logistically infeasible, I *never decline* an invitation to give a talk.

4.1 Conferences

- **Nov 2024**: 17th Conference on Informatics (Informatics 2024), Poprad, Slovakia.
  Title: "Formal Methods for Machine Learning"
- **Oct 2024**: 31st Static Analysis Symposium (SAS 2024), Pasadena, USA.
  Title: "Abstract Interpretation-Based Certification of Hyperproperties for High-Stakes Machine Learning Software"
- **Apr 2023**: 29th Symposium on Model Checking of Software (SPIN 2023), Paris, France.
  Title: "Interpretability-Aware Verification of Machine Learning Software"
- **Oct 2019**: 26th Static Analysis Symposium (SAS 2019), Porto, Portugal
  Title: "Static Analysis of Data Science Software"
  `https://youtu.be/DX_w0rq9J18`
- **Jan 2016**: Congrès SIF 2016, Strasbourg, France.
  Title: "Analyse Statique par Interprétation Abstraite de Propriétés Temporelles des Programmes"

4.2 Workshops and Working Groups

- **Oct 2024**: 10th Workshop on Numerical and Symbolic Abstract Domains (NSAD 2024), Pasadena, USA
  Title: "Abstract Domains for Machine Learning Verification"
- **Apr 2024**: 29th Journées Formalisation des Activités Concurrentes (FAC 2024), Toulouse, France
  Title: "Machine Learning Interpretability and Verification"
- **Jul 2022**: "Vistas in Verified Software" Workshop, "Verified Software" Programme, Isaac Newton Institute for Mathematical Sciences, UK (remote)
  Title: "Static Analysis for Data Scientists"
- **Jun 2022**: 11th Workshop on the State Of the Art in Program Analysis (SOAP 2022), San Diego, USA
  Title: "Static Analysis for Data Scientists"
- **May 2022**: 1st Symposium on Challenges of Software Verification (CSV 2022), Venice, Italy
  Title: "Static Analysis for Data Scientists"
- **Nov 2021**: Journées du GT Vérif 2021, ENS Paris-Saclay, Gif-sur-Yvette, France
  Title: "An Abstract Interpretation Recipe for Machine Learning Fairness"
- **Jul 2021**: 4th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS 2021), Los Angeles, USA (remote)
  Title: "An Abstract Interpretation Recipe for Machine Learning Fairness"
- **Jan 2021**: Lorentz Center Workshop "Robust Artificial Intelligence", Lorentz Center, The Netherlands (remote)
  Title: "Perfectly Parallel Fairness Certification of Neural Networks"
- **Jan 2021**: Lorentz Center Workshop "Robust Artificial Intelligence", Lorentz Center, The Netherlands (remote)

Title: "Formal Methods for Robust Artificial Intelligence: State of the Art"
https://www.youtube.com/watch?v=ayXLWs4G4RU

- **Jul 2020**: 2nd Workshop on Democratizing Software Verification (DSV 2020), Los Angeles, USA (remote)
  Title: "A Static Analyzer for Data Science Software"
  https://www.youtube.com/watch?v=f8Cjpt-rzxE&t=4374s
- **Nov 2013**: 2nd Workshop on Analysis and Verification of Dependable Cyber Physical Software (AVDCPS 2013), Changsha, China
  Title: "The Abstract Domain of Piecewise-Defined Ranking Functions"

4.3 Invitational Seminars

- **Jun 2024:** Dagstuhl Seminar 25242 "Testing Program Analyzers and Verifiers", Schloss Dagstuhl, Germany
- **Feb 2024:** Dagstuhl Seminar 25061 "Logic and Neural Networks", Schloss Dagstuhl, Germany
  Title: "Static Analysis Methods for Neural Networks"
- **Jul 2022:** Dagstuhl Seminar 22291 "Machine Learning and Logical Reasoning: The New Frontier", Schloss Dagstuhl, Germany
  Title: "Data Usage across the Machine Learning Pipeline"
- **Oct 2017:** Shonan Meeting 100 "Analysis and Verification of Pointer Programs", Shonan Village Center, Japan
  Title: "An Abstract Interpretation Framework for Input Data Usage"
- **Sep 2017:** Shonan Meeting 108 "Memory Abstraction, Emerging Techniques and Applications", Shonan Village Center, Japan
  Title: "An Abstract Interpretation Framework for Input Data Usage"
- **May 2016:** Dagstuhl Seminar 16201 "Synergies among Testing, Verification, and Repair for Concurrent Programs", Schloss Dagstuhl, Germany
  Title: "Bringing Abstract Interpretation to Termination and Beyond"
- **Aug 2014:** Dagstuhl Seminar 14352 "Next Generation Static Software Analysis Tools", Schloss Dagstuhl, Germany
  Title: "Automatic Inference of Ranking Functions by Abstract Interpretation"

4.4 Other Seminars

- **Jun 2025:** University of Parma, Parma, Italy
  Title: "Termination Resilience Static Analysis"
- **Feb 2024:** Airbus, Toulouse, France
  Title: "(Hyper)Safety Certification of Neural Network Surrogates for Aircraft Braking Distance Estimation"
- **Jun 2024:** Université de La Réunion, Saint-Denis, Réunion (remote)
  Title: "Abstract Interpretation"
- **Jun 2024:** Scientific Board Meeting, Inria Paris, Paris, France
  Title: "Formal Methods for Machine Learning ~~Verification~~"
- **Mar 2024:** Quarkslab, Paris, France
  Title: "Machine Learning Interpretability and Verification"
- **Mar 2023:** CEA-LIST, Palaiseau, France
  Title: "Interpretability-Aware Verification of Machine Learning Software"
- **Feb 2023:** Séminaire IRILL, Center for Research and Innovation on Free Software, Paris, France
  Title: "Interpretability-Aware Verification of Machine Learning Software"
- **May 2022:** La Demi-Heure de Science, Inria Paris, Paris, France
  Title: "Interprétation Abstraite des Réseaux de Neurones"

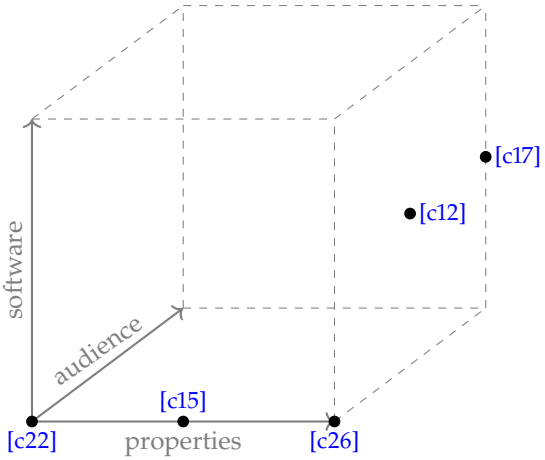
- **Nov 2021:** CEA-LIST, Palaiseau, France
  Title: "An Abstract Interpretation Recipe for Machine Learning Fairness"
- **May 2021:** École Normale Supérieure, Paris, France (remote)
  Title: "Perfectly Parallel Fairness Certification of Neural Networks"
- **Feb 2021:** Airbus, Toulouse, France (remote)
  Title: "Formal Methods for Robust Artificial Intelligence: State of the Art"
- **Nov 2020:** INSERM, Paris, France (remote)
  Title: "Static Analysis for Data Science"
- **Jun 2020:** Inria Rennes, Rennes, France (remote)
  Title: "Perfectly Parallel Fairness Certification of Neural Networks"
- **Jun 2020:** IRIF, Paris, France (remote)
  Title: "Perfectly Parallel Fairness Certification of Neural Networks"
- **May 2020:** Tel Aviv University, Tel Aviv, Israel (remote)
  Title: "Perfectly Parallel Fairness Certification of Neural Networks"
- **May 2020:** Thales Research & Technology, Palaiseau, France (remote)
  Title: "Perfectly Parallel Fairness Certification of Neural Networks"
- **Apr 2019:** Gran Sasso Science Institute (GSSI), L'Aquila, Italy
  Title: "What Programs Want: Automatic Inference of Input Data Specifications"
- **May 2018:** Inria Paris, Paris, France
  Title: "Static Program Analysis for a Software-Driven Society"
- **May 2018:** TU Wien, Vienna, Austria
  Title: "Static Program Analysis for a Software-Driven Society"
- **Mar 2018:** École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
  Title: "Static Program Analysis for a Software-Driven Society"
- **Mar 2018:** Stevens Institute of Technology, Hoboken, New Jersey, USA
  Title: "Static Program Analysis for a Software-Driven Society"
- **Mar 2018:** Max Planck Institute for Software Systems, Kaiserslautern, Germany
  Title: "Static Program Analysis for a Software-Driven Society"
- **Jan 2017:** Université Pierre et Marie Curie (Paris 6), Paris, France
  Title: "Synthesizing Ranking Functions from Bits and Pieces"
- **Aug 2015:** TU Wien, Vienna, Austria
  Title: "Abstract Interpretation as Automated Deduction"
- **Jul 2015:** SRI International, Menlo Park, USA
  Title: "Counterexample-Guided Inference of Ranking Functions"
- **Dec 2014:** University of Udine, Udine, Italy
  Title: "Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation"
- **Nov 2014:** ETH Zurich, Zurich, Switzerland
  Title: "Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation"
- **Oct 2014:** Queen Mary University of London, London, UK
  Title: "Automatic Inference of Ranking Functions by Abstract Interpretation"
- **Jun 2014:** University College London, London, UK
  Title: "Automatic Inference of Ranking Functions by Abstract Interpretation"
- **May 2014:** Inria Rennes, Rennes, France
  Title: "An Abstract Domain to Infer Ordinal-Valued Ranking Functions"
- **Mar 2014:** Inria Paris-Rocquencourt, France
  Title: "Automatic Inference of Ranking Functions by Abstract Interpretation"
- **Jan 2014:** IBM Thomas J. Watson Research Center, Yorktown Heights, USA
  Title: "Automatic Inference of Ranking Functions by Abstract Interpretation"
- **Nov 2013:** East China Normal University, Shanghai, China
  Title: "The Abstract Domain of Piecewise-Defined Ranking Functions"
- **Mar 2013:** University of Udine, Udine, Italy
  Title: "The Abstract Domain of Segmented Ranking Functions"

5

Critical Summary

Résumé critique

This section presents a critical summary of the following five scientific publications, selected for their significance within my research trajectory:



[c15] **Caterina Urban**, Samuel Ueltschi, Peter Müller. Abstract Interpretation of CTL Properties
In 25th Static Analysis Symposium (SAS 2018).
https://caterinaurban.github.io/publication/sas2018/

[c26] Naïm Moussaoui Remil, **Caterina Urban**, Antoine Miné. Automatic Detection of Vulnerable Variables for CTL Properties of Programs.
In 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2024).
https://inria.hal.science/hal-04710215

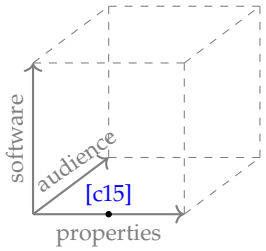
[c12] **Caterina Urban**, Peter Müller. An Abstract Interpretation Framework for Input Data Usage.
In 27th European Symposium on Programming (ESOP2018).
https://caterinaurban.github.io/publication/esop2018/

[c17] **Caterina Urban**, Maria Christakis, Valentin Wüstholz, Fuyuan Zhang. Perfectly Parallel Fairness Certification of Neural Networks.
In ACM on Programming Languages (PACMPL), Conference on Object-Oriented Programming Systems, Languages, and Applications 2020 (OOPSLA 2020).
https://inria.hal.science/hal-03091870

[c22] Marco Campion, **Caterina Urban**, Mila Dalla Preda, Roberto Giacobazzi. A Formal Framework to Measure the Incompleteness of Abstract Interpretations.
In 30th Static Analysis Symposium (SAS 2023).
https://inria.hal.science/hal-04249990

Each publication is contextualized and accompanied by a critical reflection on the scientific choices made, the results obtained and their impact, the limitations encountered, as well as the research questions raised and the directions opened for future work.

5.1 Abstract Interpretation of CTL Properties



[c15] Urban et al - Abstract Interpretation of CTL Properties (SAS 2018)

Computation tree logic (CTL) is a temporal logic introduced by Clarke and Emerson to overcome certain limitations of linear temporal logic (LTL) for program specification purposes. *With [c15], I substantially generalized my doctoral thesis*, introducing a static analysis framework based on abstract interpretation for verifying functional properties of programs expressed in CTL. This work addresses limitations of previous approaches in the literature which were often restricted to finite-state programs (or only certain classes of infinite-state program), or limited in scope to subsets of CTL without existential quantifiers (or only supporting an indirect treatment of existential quantifiers).

Central to the approach is the abstraction of the operational semantics of a given program into a function mapping program states to ordinals, bounding the number of program execution steps needed to satisfy a given CTL formula. This semantics is defined inductively on the structure of the CTL formula, enabling a principled and *compositional* treatment of functional program properties with arbitrary alternations of universal and existential quantifiers, and unifying reasoning about safety and liveness properties in a single formalism, i.e., leveraging ranking functions for liveness properties. Further decidable approximations are derived by leveraging the abstract domain based on *piecewise-defined functions* [c4] developed as part of my doctoral thesis, augmented with *under-approximating operators* [Miné12] to handle CTL formulas with existential quantifiers. A piecewise-defined function for a CTL formula is automatically inferred through backward static analysis of the given program by building upon the piecewise-defined functions for its subformulas. It over-approximates the value of the corresponding concrete semantics and, by under-approximating its domain of definition, yields a *sufficient precondition* for the CTL formula. The analysis thus provides actionable insights even when properties are only conditionally satisfied.

[c4] Urban and Miné - A Decision Tree Abstract Domain for Proving Conditional Termination (SAS 2014)
[Miné12] Miné - Inferring Sufficient Conditions with Backward Polyhedral Under-Approximations (2012)

Let us consider the program snippet in Figure 5.1 and the CTL formula $\mathsf{AG}(x = 1 \Rightarrow \mathsf{AF}(x = 0))$ stating that it is always the case (AG) that whenever the lock is acquired ($x = 1$) it will always eventually (AF) be released ($x = 0$). The static analysis proposed in [c15] automatically infers the following piecewise-defined ranking function at program point 4:

$$\lambda x n.\ \begin{cases} 0 & x = 0 \\ 2 & x \neq 0 \wedge n \leq 0 \\ 2n + 2 & \text{otherwise} \end{cases}$$

Its value indicates the maximum number of program execution steps needed to reach the next state where the lock is released, i.e., the next state that satisfies $x = 0$. The function inferred at the beginning of the program, program point 1, is only defined when $x \neq 1$ (i.e., the lock is not

```
while [1]( rand() ) {
  [2]x := 1
  [3]n := rand()
  while [4]( n > 0 ) {
    [5]n := n − 1
  }
  [6]x := 0
}
while [7]( true ) {}[8]
```

Figure 5.1: Standard lock acquire/release-style program [Cook12], where rand() is a random number generation function. Assignments x := 1 and x := 0 are acting as acquire and release, respectively

[Cook12] Cook et al - Temporal Property Verification as a Program Analysis Task (2012)

already acquired initially), yielding the weakest sufficient precondition for the CTL formula $\mathsf{AG}(x = 1 \Rightarrow \mathsf{AF}(x = 0))$.

The approach is implemented in FuncTion and the experimental evaluation in [c15] showed its effectiveness on a wide variety of benchmarks. Nonetheless, the precision of the static analysis is sensitive to the employed widening heuristic [c11], and can degrade considerably due to unfortunate interactions between under-approximations needed for existential CTL formulas and non-deterministic variable assignments.

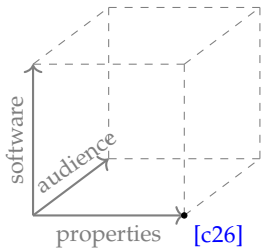
[c11] Courant and Urban - Precise Widening Operators for Proving Termination by Abstract Interpretation (TACAS 2017)
[Rival07] Rival and Mauborgne - The Trace Partitioning Abstract Domain (2007)

The framework is expressive enough to support further extensions toward LTL or CTL*. However, accommodating these logics would require the integration of some form of trace partitioning [Rival07], since the interpretation of LTL formulas is defined in terms of program executions rather than program states as CTL.

This work served as a starting point and building block for the Ph.D. thesis of Naïm Moussaoui Remil. This is further discussed in the next section.

5.2 Automatic Detection of Vulnerable Variables for CTL Properties of Programs

This work departs from the usual setting in which all program variables are treated equally, and instead adopts a refined perspective that distinguishes between variables that are *under the control of an external adversary* and those that remain *uncontrolled*. Such distinction was considered by earlier work [Girol21, Parolini24] but limited to the analysis of safety program properties, while [c26] extends this refined perspective to a much broader class of functional program properties. Specifically, [c26] proposes a static analysis that automatically identifies sets of vulnerable program variables, that is, subset-*minimal sets of program variables that need to be controlled* to ensure the satisfaction of a functional program property expressed in CTL. The analysis is a principled way to reason about attacker capabilities, agnostic to the underlying functional verification method, and [c26] presents an instance built on top of the static analysis framework for CTL discussed in the previous section [c15]. Thus, as a by-product, the approach also infers sufficient preconditions over the vulnerable variables that guarantee the satisfaction of a given CTL formula.



[c26] Moussaoui Remil et al. - Automatic Detection of Vulnerable Variables for CTL Properties of Programs (LPAR 2024)

[Girol21] Girol et al. - Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference
[Parolini24] Parolini and Miné - Sound Abstract Nonexploitability Analysis (VMCAI 2024)

Let us consider the program snippet on the right and the CTL formula $\mathsf{AF}(3 : \text{true})$ stating that the error location at program point 3 is always eventually (AF) reached. The static analysis proposed in [c26] automatically infers that $\{y, z\}$ and $\{x\}$ are alternative sets of vulnerable variables: indeed, it is enough to control the value of y and z (such that $y \leq z$ and the loop is never entered), or to control the value of x (such that $x \geq 0$ and the loop always terminates) to ensure the reachability of the error location *independently of the values of the uncontrolled variables*.

```
while 1( y > z ) {
  2y := y - z
}
3 // error
```

This work constitutes the first contribution made by my Ph.D. student Naïm Moussaoui Remil. He is currently addressing the limitation of the approach to numerical programs without pointers or memory manipulations. Specifically, he is developing an extension of the underlying static analysis framework for CTL [c15] to handle pointer-manipulating programs. This enhancement is essential for broadening the applicability of the approach proposed in [c26] to more realistic and complex programs,

notably in security, where vulnerable variable sets would identify attack vectors that could compromise a system.

More fundamentally, the distinction between controlled and uncontrolled variables has deep semantic implications, as it enables the coexistence of different flavors of non-determinism within the same program: demonic non-determinism, associated with adversarial control, and angelic non-determinism, reflecting benign or arbitrary behavior. This gives rise to interesting program hyperproperties such as *termination resilience* – the impossibility for an adversary to cause definite non-termination of a program independently of the value of the uncontrolled variables [u4] – and *functional non-exploitability* – the impossibility for an adversary to cause a program to violate or satisfy a functional property (generalizing safety non-exploitability [Parolini24]).

The identification of vulnerable variables can be interpreted as a form of causal attribution – highlighting which program variables are responsible for the satisfaction or violation of a given property under adversarial influence. This opens promising connections with *causality* and *responsibility analysis* [Deng19] that are worth exploring. It also aligns with current trends in logic-based explainability, where formal reasoning techniques are used to provide minimal sufficient explanations of the behavior of a system [Marques-Silva24]. In this light, vulnerable variable sets and their associated preconditions offer interpretable evidence for understanding why a program exhibits a certain behavior.

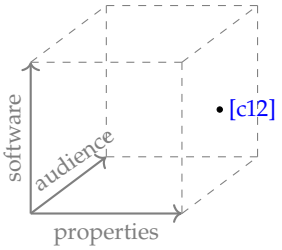
[u4] Moussaoui Remil and Urban - Termination Resilience Static Analysis
[Parolini24] Parolini and Miné - Sound Abstract Nonexploitability Analysis (VMCAI 2024)
[Deng19] Deng and Cousot - Responsibility Analysis by Abstract Interpretation (SAS 2019)
[Marques-Silva24] Marques-Silva - Logic-Based Explainability: Past, Present and Future (ISoLA 2024)

5.3 An Abstract Interpretation Framework for Input Data Usage

This work represents my very first step in *democratizing static analysis* towards a different kind of software programs – *data science software* – and targeting a different audience – *data scientists*. It was motivated by the pervasiveness of data-driven decision-making software in many domains, ranging from retail, to manufacturing, finance, and even healthcare.

In [c12], I focused on leveraging static program analysis to detect program flaws that do not cause failures. Such silent mistakes can have serious consequences since code that produces an erroneous but plausible result gives no indication that something went wrong. In particular, I proposed an abstract interpretation framework to automatically detect input *data that remains unused* by a program.

Central to the framework is a characterization of when a program uses (some of) its input data by means of a notion of *dependency* between the input data and the outcome of the program. Its definition, shown in Figure 5.2, formalizes the idea that a program input i is used if a certain outcome of the program (σ_ω) is not possible when the input variable i has a certain value (v). Unlike the usual definitions – notably in the secure information flow literature – this definition formalizes that changing the value of i affects the outcome of the program in a way that also accounts for non-termination (since the outcome σ_ω is either a final state or non-termination) as well as non-determinism (via the universal quantification over all program execution traces σ' where i has initial value v).



[c12] Urban and Müller - An Abstract Interpretation Framework for Input Data Usage (ESOP 2018)

$$\textsc{used}_i \stackrel{\text{def}}{=} \exists \sigma v \colon A \wedge \forall \sigma' \colon B \Rightarrow C$$
$$A \stackrel{\text{def}}{=} \sigma_0(i) \neq v$$
$$B \stackrel{\text{def}}{=} \sigma_0 \equiv_{\setminus i} \sigma'_0 \wedge \sigma'_0(i) = v$$
$$C \stackrel{\text{def}}{=} \sigma_\omega \neq \sigma'_\omega$$

Figure 5.2: Definition of when a program uses an input variable i, where σ, σ' are execution traces of the program, v is a value for i, σ_0 and σ_ω represent the initial state of a trace and its outcome (a final state or non-termination), and $\sigma_0 \equiv_{\setminus i} \sigma'_0$ denotes initial states of program traces only differing on the value of i.

This key definition encompasses notions of dependencies that arise in many different contexts, such as secure information flow, program slicing, and provenance or lineage analysis. It thus provides a *unifying framework* for reasoning about existing analyses based on dependencies and their applicability to detect unused input data. In [c12], I surveyed non-interference [Assaf17] and strongly-live variable [Giegerich81] analyses and identified key design decisions that hinder or facilitate their applicability. I additionally proposed a more precise static analysis based on *syntactic dependencies* between program variables, accompanied by an implementation targeting Python programs in the open-source tool Lyra.

[Assaf17] Assaf et al. - Hypercollecting Semantics and Its Application to Static Analysis of Information Flow (POPL 2017)
[Giegerich81] Giegerich et al. - Invariance of Approximate Semantics with Respect to Program Transformations (1981)

This foundational work has catalyzed a large body of subsequent research, spanning both theoretical developments and practical applications.

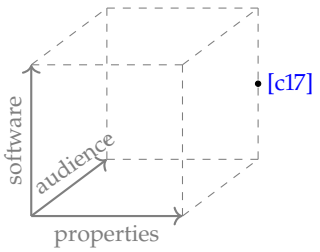
First of all, it led to the Ph.D. thesis of Denis Mazzucato, which introduced a *quantitative generalization* of input data usage aimed at assessing the extent to which a program input influences its behavior [c25]. One notable instance of this quantitative framework focused on the impact of program inputs on the global number of program loop iterations [c28]. Its practical implementation, building upon the syntactic dependency analysis proposed in [c12], enabled proving the immunity to timing side-channel attacks of the real-world cryptographic library S2N-Bignum.

[c25] Mazzucato et al. - Quantitative Input Usage Static Analysis (NFM 2024)
[c27] Drobnjakovic et al. - An Abstract Interpretation-Based Data Leakage Static Analysis (TASE 2024)
[c28] Mazzucato et al. - Quantitative Static Timing Analysis (SAS 2024)
[w5] Dolcetti et al. - Towards a High Level Linter for Data Science (NSAD 2024)

Other practical applications of [c12] emerged from instances and generalization of the input-outcome dependency definition in Figure 5.2. An instance applied to neural networks enabled proving *absence of algorithmic bias*, and is further discussed in the next section. A generalization to multi-dimensional program variables led to an abstract interpretation framework for proving *absence of data leakage* between data used for training and testing machine learning models [c27].

Moreover, ongoing work on an abstract interpretation-based linter tool for data science practitioners [w5], builds upon Lyra.

5.4 Perfectly Parallel Fairness Certification of Neural Networks



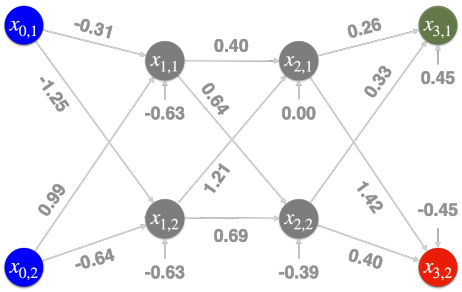
A number of cases over the years have shown that machine-learned systems may reproduce or exacerbate bias explicitly or implicitly embedded in their training data [Larson16, Obermeyer19]. In response to these concerns, the European Commission introduced the Artificial Intelligence Act in April 2021, which establishes a comprehensive legal framework to govern the development and deployment of machine learning systems, with particular emphasis on preventing discriminatory outcomes. Within this regulatory landscape, [c17] proposes a valuable static analysis for assessing fairness of neural network classifiers.

[c17] Urban et al. - Perfectly Parallel Fairness Certification of Neural Networks (OOPSLA 2020)

The approach is tailored for *dependency fairness* [Galhotra17], a form of global robustness requiring the neural network classification to be independent of the values of the chosen sensitive input features. Formally, the definition of (dependency) bias with respect to a sensitive feature i, shown in Figure 5.3, formalizes the idea that the neural network can yield different predictions (σ_ω and σ'_ω) for input data that only differs in the value of i. It is an instance of the definition in Figure 5.2, simplified since trained neural networks are deterministic and always terminating.

[Larson16] Larson et al. - How We Analyzed the COMPAS Recidivism Algorithm (2016)
[Obermeyer19] Obermeyer et al. - Dissecting Racial Bias in an Algorithm Used to Manage the Health of Populations (2019)

Let us consider the following toy neural network binary classifier:



It comprises two input neurons $x_{0,1}$ (credit amount) and $x_{0,2}$ (age of the person requesting the credit), and two output neurons $x_{3,1}$ (request approved) and $x_{3,2}$ (request denied). In between there are two hidden layers, each with two hidden neurons. Each hidden or output neuron computes an affine combination of the values of the neurons in the previous layer using the weights on the edges. Hidden neurons additionally apply an activation function such as the ReLU (i.e., ReLU(x) = max(0, x)). We say that the neuron is active when $x > 0$ before applying the ReLU, and inactive when $x \leq 0$. The prediction of the neural network for given values of the input neurons is the output neuron with the maximum value. Assuming that input data is normalized in the range 0 to 1, the static analysis approach proposed in [c17] automatically infers that the neural network discriminates with respect to the age of the person if the requested credit amount is larger than 0.53, which amounts to ~47% of the entire input space of the neural network.

$$\text{BIAS}_i \stackrel{\text{def}}{=} \exists \sigma \sigma' : B \Rightarrow C$$

$$B \stackrel{\text{def}}{=} \sigma_0 \equiv_{\setminus i} \sigma'_0$$

$$C \stackrel{\text{def}}{=} \sigma_\omega \neq \sigma'_\omega$$

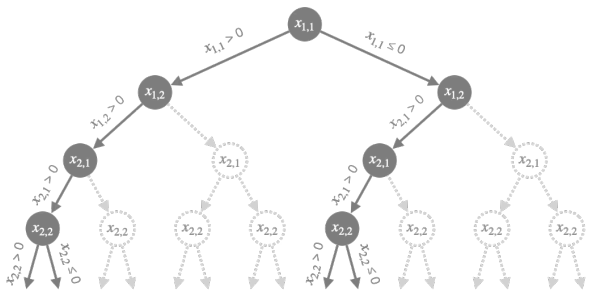
Figure 5.3: Definition of when a neural network is biased with respect to a sensitive input feature i, where σ_0 and σ_ω are the input data fed to the neural network and its prediction, and $\sigma_0 \equiv_{\setminus i} \sigma'_0$ denotes input data only differing on i.

[Galhotra17] Galhotra et al. - Fairness Testing: Testing Software for Discrimination (FSE 2017)

The approach is a clever *combination of a forward and a backward analyses* that enables perfect parallelization and a reduction in the overall analysis effort. At its core, the forward analysis employs an abstract domain to over-approximate the set of possible values that each neuron can take, and iteratively divides the input space of the neural network into independent partitions that satisfy the configured resource limits. These constrain the size of a partition with a lower bound L, and set an upper bound U on the number of hidden neurons with unknown activation status (neither always active or always inactive) found by the analysis of the partition. It turns out that, often, *multiple input partitions are associated to the same activation pattern*. In the toy example above, both partitions I_1 and $I_{2,1}$ (cf. the note on the right) have $x_{1,1}$ and $x_{2,2}$ with unknown activation status, as well as $x_{1,2}$ and $x_{2,1}$ always active.

A forward analysis of the toy neural network above with the interval abstract domain (tracking the range of possible values for each neuron) and constrained by a lower bound of 0.25 and an upper bound of 2 iteratively partitions the input space I ($x_{0,1} \in [0, 1]$ and $x_{0,2} \in [0, 1]$) of the neural network into I_1 ($x_{0,1} \in [0, 0.5]$ and $x_{0,2} \in [0, 1]$), $I_{2,1}$ ($x_{0,1} \in [0.5, 0.75]$ and $x_{0,2} \in [0, 1]$) and $I_{2,2}$ ($x_{0,1} \in [0.75, 1]$ and $x_{0,2} \in [0, 1]$). The analysis finds that all hidden neurons have unknown activations status for I – then split into I_1 and I_2 ($x_{0,1} \in [0.5, 1]$ and $x_{0,2} \in [0, 1]$) – and I_2 – then split into $I_{2,1}$, $I_{2,2}$.

The backward analysis uses the abstract domain of polyhedra (tracking conjunctions of linear constraints over the neurons) to identify the subregions of the input partitions that correspond to each class prediction. It leverages the activation patterns to prune unfeasible executions.



The possible classification outcomes of the toy neural network above are represented by the constraints $x_{3,1} > x_{3,2}$ (credit request approved) and $x_{3,2} > x_{3,1}$ (credit request denied). For the input partitions I_1 and $I_{2,1}$, the analysis of the hidden layers can leverage their activation pattern to prune away the dotted execution paths on the left (since $x_{1,2}$ and $x_{2,1}$ are always active).

Finally, each identified subregion havocs any constraint on the sensitive input features and the analysis check for intersections between subregions corresponding to different class predictions. Any non-empty intersection is a *witness* of bias of the neural network: the input data in the intersection is classified differently depending on the value of the sensitive input features. If no intersection can be found then all identified subregions are certified to be fair. The analysis of the toy example above, is able to certify I_1 to be fair and, instead, finds bias in $I_{2,1}$, for $x > 0.53$, and in $I_{2,2}$.

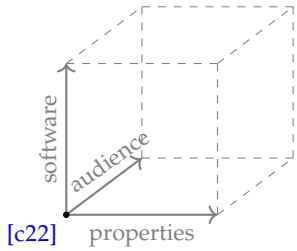
The approach is implemented in the open-source tool Libra. In follow-up work, I proposed improvements to the forward analysis. Together with my Ph.D. student Denis Mazzucato, we obtained over 10% of improvement in precision with a *reduced product* abstract domain [c18]. With my Ph.D. student Serge Durant, we improved scalability with a *smarter partitioning heuristic* that leverages the underlying abstract domain to decide how a partition is divided [w3]. This heuristic yielded a reduction of 1 to 2 orders of magnitude in the number of input partitions produced by the analysis, significantly decreasing its computational cost.

[c18] Mazzucato and Urban - Reduced Products of Abstract Domains for Fairness Certification of Neural Networks (SAS 2021)
[w3] Durand et al. - ReCIPH: Relational Coefficients for Input Partitioning Heuristic (WFVML 2022)

While scalability remains limited to neural networks with low dimensional inputs and hundreds of hidden neurons, the approach is nevertheless sufficiently effective to handle real-world applications. In a collaboration with Airbus, I successfully used the static analysis proposed in [c17] and improved in [c18, w3] to certify dependency fairness of a neural network surrogate for aircraft braking distance estimation (e.g., proving that estimating the braking distance in the expected altitude range does not alter the predicted runway overrun risk).

5.5 A Formal Framework to Measure the Incompleteness of Abstract Interpretations

This work tackles the challenge of reasoning rigorously about *incompleteness* in abstract interpretation, the inevitable loss of precision caused by the abstraction [Giacobazzi15]. It propose a formal framework for quantitatively measuring imprecision – thus enabling a principled way to reason about and compare incomplete static analyses.



[c22]

[c22] Campion et al. - A Formal Framework to Measure the Incompleteness of Abstract Interpretations (SAS 2023)

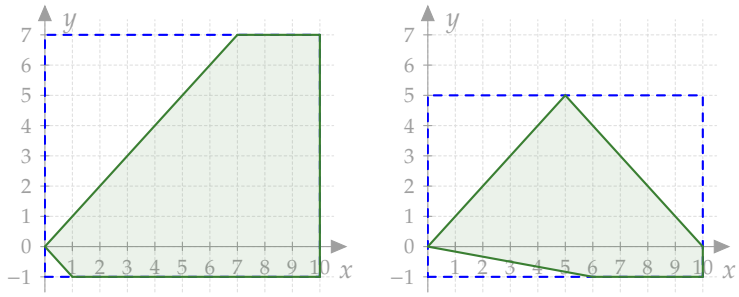
In particular, [c22] generalizes the notion of *partial completeness* introduced by [Campion22] using *pre-metrics* compatible with the partial order underlying the abstraction. Let $\langle C, \preccurlyeq \rangle$ be a partially ordered set. A pre-metric $\delta \colon C \times C \to \mathbb{R} \cup \{\infty\}$ is non-negative ($\forall x, y \in C \colon \delta(x, y) \geq 0$) and satisfies the if-identity axiom ($\forall x, y \in C \colon x = y \Rightarrow \delta(x, y) = 0$). It is compatible with $\langle C, \preccurlyeq \rangle$ if it is meaningful for comparing elements on the same chain in C, i.e., it additionally satisfies $\forall x, y, z \in C \colon x \preccurlyeq y \preccurlyeq z \Rightarrow \delta(x, y) \leq \delta(x, z) \wedge \delta(y, z) \leq \delta(x, z)$. Pre-metrics broaden the applicability of partial completeness to concretization-based abstractions, while the original definition in [Campion22] is limited by the requirement of having a Galois connection. The generalized definition of partial completeness is shown in Definition 5.5.1. The distance δ between $f \circ \gamma$ and $\gamma \circ f^\natural$ is a measure of the imprecision introduced by $f^\natural$ with respect to f, and the definition requires this imprecision to be bounded by ε.

[Giacobazzi15] Giacobazzi et al. - Analyzing Program Analyses (POPL 2015)
[Campion22] Campion et al. - Partial (In)Completeness in Abstract Interpretation: Limiting the Imprecision in Program Analysis (POPL 2022)

Definition 5.5.1 (Partial Completeness) *Given pre-ordered sets $\langle C, \preccurlyeq \rangle$ and $\langle A, \leq \rangle$ related by a concretization function $\gamma \colon C \to A$ and a pre-metric δ compatible with $\langle C, \preccurlyeq \rangle$, a function $f^\natural \colon A \to A$ is an* ε-partial complete approximation *of a function $f \colon C \to C$ if and only if $\forall x \in A \colon \delta(f(\gamma(x)), \gamma(f^\natural(x))) \leq \varepsilon$.*

Let us consider the program P in Figure 5.4. The best linear convex approximation of the loop invariant at program point 2 is $0 \leq y \leq x \wedge x + y \leq 10$

represented by the green triangle in Figure 5.5. We measure the imprecision introduced by a numerical static analysis with the difference in percentage between the volumes of the hyperrectangles enclosing the loop invariants (shown in dashed blue for the best invariant in Figure 5.5). These are the loop invariants inferred by INTERPROC using the octagons [Mine01] (left) and polyhedra [Cousot78] (right) abstract domains:



The static analysis using octagons is 60-partial complete: the hyperrectangle enclosing the inferred invariant has 60% more volume than the one in Figure 5.5. Instead, the analysis using polyhedra is 20-partial complete. The generality of the framework proposed [c22] also allows measuring the relative precision of static analyses: in this case, the analysis using octagons is 33.33-partial complete with respect to using polyhedra.

This work led to a fruitful ongoing collaboration centered on the study of novel interesting hyperproperties that arise from combining qualitative approximations – modeled using upper closure operators [Cousot79] – and quantitative approximations – expressed via pre-metrics. We are currently studying *abstract Lipschitz continuity* [u2] – a generalization of Lipschitz continuity that ensures that small differences in *semantic approximations* of inputs to a function (e.g., a program) lead to proportionally bounded differences in the *semantic approximations* of its outputs – and *partial abstract non-interference* [u3] – a generalization of abstract non-interference [Giacobazzi04] that admits a bounded error in output.

```
[1] x := 0
[2] y := 0
while [2]( x ≤ 9 ∧ y ≥ 0 ) {
  [3]if ( x ≤ 4 ) {
    [4]y := y + 1
  } else {
    [6]y := y - 1
  }
  [6]x := x + 1
}
```

Figure 5.4: The program P [c22].

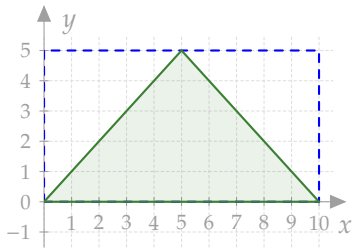


Figure 5.5: The best convex approximation of the loop invariant of program P.

[Mine01] Miné - A New Numerical Abstract Domain Based on Difference-Bound Matrices (PADO 2021)
[Cousot78] Cousot and Halbwachs - Automatic Discovery of Linear Restraints Among Variables of a Program (POPL 1978)
[Cousot79] Cousot and Cousot - Systematic Design of Program Analysis Frameworks (POPL 1979)
[Giacobazzi04] Giacobazzi and Mastroeni - Abstract Non-Inteference: Parametrizing Non-Interference by Abstract Interpretation (POPL 2004)
[u2] Campion et al. - Abstract Lipschitz Continuity
[u3] Campion et al. - Measuring vs Abstracting: On the Relation between Distances and Abstract Domains

Habilitation to Supervise Research (HDR) Application File

Caterina Urban

Scientific Work Sample

Exemplaire des travaux scientifiques

[c15] **CU**, Samuel Ueltschi, Peter Müller

Abstract Interpretation of CTL Properties

In 25th Static Analysis Symposium (SAS 2018)

https://caterinaurban.github.io/publication/sas2018/

Abstract Interpretation of CTL Properties

Caterina Urban( ), Samuel Ueltschi, and Peter Müller

Department of Computer Science, ETH Zurich, Zurich, Switzerland
caterina.urban@inf.ethz.ch

Abstract. CTL is a temporal logic commonly used to express program properties. Most of the existing approaches for proving CTL properties only support certain classes of programs, limit their scope to a subset of CTL, or do not directly support certain existential CTL formulas. This paper presents an abstract interpretation framework for proving CTL properties that does not suffer from these limitations. Our approach automatically infers sufficient preconditions, and thus provides useful information even when a program satisfies a property only for some inputs. We systematically derive a program semantics that precisely captures CTL properties by abstraction of the operational trace semantics of a program. We then leverage existing abstract domains based on piecewise-defined functions to derive decidable abstractions that are suitable for static program analysis. To handle existential CTL properties, we augment these abstract domains with under-approximating operators. We implemented our approach in a prototype static analyzer. Our experimental evaluation demonstrates that the analysis is effective, even for CTL formulas with non-trivial nesting of universal and existential path quantifiers, and performs well on a wide variety of benchmarks.

1 Introduction

Computation tree logic (CTL) [6] is a temporal logic introduced by Clarke and Emerson to overcome certain limitations of linear temporal logic (LTL) [33] for program specification purposes. Most of the existing approaches for proving program properties expressed in CTL have limitations that restrict their applicability: they are limited to finite-state programs [7] or to certain classes of infinite-state programs (e.g., pushdown systems [36]), they limit their scope to a subset of CTL (e.g., the universal fragment of CTL [11]), or support existential path quantifiers only indirectly by considering their universal dual [8].

In this paper, we propose a new static analysis method for proving CTL properties that does not suffer from any of these limitations. We set our work in the framework of *abstract interpretation* [16], a general theory of semantic approximation that provides a basis for various successful industrial-scale tools

A. Podelski (Ed.): SAS 2018, LNCS 11002, pp. 402–422, 2018.
https://doi.org/10.1007/978-3-319-99725-4_24

```
while [1]( rand() ) {
  [2]x := 1
  [3]n := rand()
  while [4]( n > 0 ) { [5]n := n − 1 }
  [6]x := 0
}
while [7]( true ) {}[8]
```

Fig. 1. Standard lock acquire/release-style program [12], where rand() is a random number generation function. Assignments $x := 1$ and $x := 0$ are acting as acquire and release, respectively. We want to prove the CTL property $\mathsf{AG}(x = 1 \Rightarrow \mathsf{A}(true\ \mathsf{U}\ x = 0))$ expressing that whenever a lock is acquired ($x = 1$) it is eventually released ($x = 0$). We assume that initially $x = 0$.

(e.g., Astrée [3]). We generalize an existing abstract interpretation framework for proving termination [18] and other liveness properties [41].

Following the theory of abstract interpretation [14], we abstract away from irrelevant details about the execution of a program and systematically derive a program semantics that is *sound and complete* for proving a CTL property. The semantics is a function defined over the programs states that satisfy the CTL formula. The value of the semantics for a CTL formula that expresses a liveness property (e.g., $\mathsf{A}(true\ \mathsf{U}\ \phi)$) gives an upper bound on the number of program execution steps needed to reach a desirable state (i.e., a state satisfying ϕ for $\mathsf{A}(true\ \mathsf{U}\ \phi)$). The semantics for any other CTL formula is the constant function equal to zero over its domain. We define the semantics inductively on the structure of a CTL formula, and we express it in a constructive fixpoint form starting from the functions defined for its sub-formulas.

Further sound abstractions suitable for static program analysis are derived by *fixpoint approximation* [14]. We leverage existing numerical abstract domains based on piecewise-defined functions [39], which we augment with novel under-approximating operators to directly handle existential CTL formulas. The piecewise-defined function for a CTL formula is automatically inferred through *backward analysis* by building upon the piecewise-defined functions for its sub-formulas. It over-approximates the value of the corresponding concrete semantics and, by under-approximating its domain of definition, yields a *sufficient precondition* for the CTL property. We prove the soundness of the analysis, meaning that all program executions respecting the inferred precondition indeed satisfy the CTL property. A program execution that does not respect the precondition might or might not satisfy the property.

To briefly illustrate our approach, let us consider the acquire/release-style program shown in Fig. 1, and the CTL formula $\mathsf{AG}(x = 1 \Rightarrow \mathsf{A}(true\ \mathsf{U}\ x = 0))$. The analysis begins from the atomic propositions $x = 1$ and $x = 0$ and, for each program control point, it infers a piecewise-defined function that is only defined when x is one or zero, respectively. It then continues to the sub-formula

$\mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$ for which, building upon the function obtained for $\mathrm{x} = 0$, it infers the following interesting function at program point 4:

$$\lambda \mathrm{x}.\lambda \mathrm{n}.\begin{cases} 0 & \mathrm{x} = 0 \\ 2 & \mathrm{x} \neq 0 \wedge \mathrm{n} \leq 0 \\ 2\mathrm{n} + 2 & \text{otherwise} \end{cases} \tag{1.1}$$

The function indicates that the sub-formula $\mathrm{x} = 0$ is either satisfied trivially (when x is already zero), or in at most 2 program execution steps when $\mathrm{n} \leq 0$ (and thus the loop at program point 4 is not entered) and $2\mathrm{n} + 2$ steps when $\mathrm{n} > 0$ (and thus the loop is entered). The analysis then proceeds to $\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$, i.e., $\mathrm{x} \neq 1 \vee \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$. The inferred function for the sub-formula $\mathrm{x} \neq 1$ is only defined over the complement of the domain of the one obtained for $\mathrm{x} = 1$. The disjunction combines this function with the one obtained for $\mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$ by taking the union over the function domains and the maximum over the function values. The result at program point 4 is the same function obtained for $\mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0)$. Finally, the analysis can proceed to the initial formula $\mathsf{AG}(\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0))$. The function at program point 4 remains the same but its value now indicates the maximum number of steps needed until the *next state* that satisfies $\mathrm{x} = 0$. The function inferred at the beginning of the program proves that the program satisfies the CTL formula $\mathsf{AG}(\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0))$ unless x has initial value one. Indeed, in such a case, the program does not satisfy the formula since the loop at program point 1 might never execute. Thus, the inferred precondition is the weakest precondition for the CTL property $\mathsf{AG}(\mathrm{x} = 1 \Rightarrow \mathsf{A}(\mathit{true}\ \mathsf{U}\ \mathrm{x} = 0))$.

We implemented our approach in the prototype static analyzer FuncTion [13]. Our experimental evaluation demonstrates that the analysis is effective, even for CTL formulas with non-trivial nesting of universal and existential path quantifiers, and performs well on a wide variety of benchmarks.

2 Trace Semantics

We model the operational semantics of a program as a *transition system* $\langle \Sigma, \tau \rangle$ where Σ is a (potentially infinite) set of program states, and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the possible transitions between states. The set of *final states* of the program is $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma \colon \langle s, s' \rangle \notin \tau\}$.

Given a transition system $\langle \Sigma, \tau \rangle$, the function $\mathrm{pre} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a given set of states X to the set of their predecessors with respect to τ:$\mathrm{pre}(X) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \exists s' \in X : \langle s, s' \rangle \in \tau\}$, and the function $\widetilde{\mathrm{pre}} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a given set of states X to the set of states whose successors with respect to τ are all in X: $\widetilde{\mathrm{pre}}(X) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \in \tau \Rightarrow s' \in X\}$.

In the following, given a set S, let $S^n \stackrel{\text{def}}{=} \{s_0 \cdots s_{n-1} \mid \forall i < n : s_i \in S\}$ be the set of all sequences of exactly n elements from S. We write ε to denote the empty sequence, i.e., $S^0 \stackrel{\text{def}}{=} \{\varepsilon\}$. Let $S^* \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} S^n$ be the set of all finite

sequences, $S^+ \stackrel{\text{def}}{=} S^* \setminus S^0$ be the set of all non-empty finite sequences, S^ω be the set of all infinite sequences, $S^{+\infty} \stackrel{\text{def}}{=} S^+ \cup S^\omega$ be the set of all non-empty finite or infinite sequences and $S^{*\infty} \stackrel{\text{def}}{=} S^* \cup S^\omega$ be the set of all finite or infinite sequences of elements from S. We write $\sigma\sigma'$ for the concatenation of two sequences $\sigma, \sigma' \in S^{*\infty}$ (with $\sigma\varepsilon = \varepsilon\sigma = \sigma$, and $\sigma\sigma' = \sigma$ if $\sigma \in S^\omega$), $T^+ \stackrel{\text{def}}{=} T \cap S^+$ for the selection of the non-empty finite sequences of $T \subseteq S^{*\infty}$, $T^\omega \stackrel{\text{def}}{=} T \cap S^\omega$ for the selection of the infinite sequences of $T \subseteq S^{*\infty}$, and $T \,;\, T' \stackrel{\text{def}}{=} \{\sigma s \sigma' \mid s \in S, \sigma s \in T, s\sigma' \in T'\}$ for the merging of sets of sequences $T \subseteq S^+$ and $T' \subseteq S^{+\infty}$, when a finite sequence in T terminates with the initial state of a sequence in T'.

Given a transition system $\langle \Sigma, \tau \rangle$, a *trace* is a non-empty sequence of program states described by the transition relation τ, that is, $\langle s, s' \rangle \in \tau$ for each pair of consecutive states $s, s' \in \Sigma$ in the sequence. The set of final states Ω and the transition relation τ can be understood as sets of traces of length one and two, respectively. The *maximal trace semantics* $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition system is the union of all non-empty finite traces that are terminating with a final state in Ω, and all infinite traces. It can be expressed as a least fixpoint in the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$ [14]:

$$\Lambda = \mathrm{lfp}^{\sqsubseteq}\, \lambda T. \Omega \cup (\tau \,;\, T) \tag{2.1}$$

where the computational order is $T_1 \sqsubseteq T_2 \stackrel{\text{def}}{=} T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$.

The maximal trace semantics carries all information about a program and fully describes its behavior. However, reasoning about a particular property of a program is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. In the paper, we use *abstract interpretation* [16] to systematically derive, by abstraction of the maximal trace semantics, a sound and complete semantics that precisely captures exactly and only the needed information to reason about CTL properties.

3 Computation Tree Logic

CTL is also known as *branching* temporal logic; its semantics is based on a branching notion of time: at each moment there may be several possible successor program states and thus each moment of time might have several different possible futures. Accordingly, the interpretation of CTL formulas is defined in terms of program states, as opposed to the interpretation of LTL formulas in terms of traces. This section gives a brief introduction into the syntax and semantics of CTL. We refer to [1] for further details.

We assume a set of atomic propositions describing properties of program states. Formulas in CTL are formed according to the following grammar:

$$\phi ::= a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{AX}\phi \mid \mathsf{AG}\phi \mid \mathsf{A}(\phi\ \mathsf{U}\ \phi) \mid \mathsf{EX}\phi \mid \mathsf{EG}\phi \mid \mathsf{E}(\phi\ \mathsf{U}\ \phi)$$

where a is an atomic proposition. The universal quantifier (denoted A) and the existential quantifier (denoted E) allow expressing properties of *all* or *some*

traces that start in a state. In the following, we often use Q to mean either A or E. The *next* temporal operator (denoted X) allows expressing properties about the next program state in a trace. The *globally* operator (denoted G) allow expressing properties that should hold always (i.e., for all states) on a trace. The *until* temporal operator (denoted U) allows expressing properties that should hold eventually on a trace, and always until then. We omit the *finally* temporal operator (denoted F) since a formula of the form $\mathsf{QF}\phi$ can be equivalently expressed as $\mathsf{Q}(\mathit{true}\ \mathsf{U}\ \phi)$.

The semantics of formulas in CTL is formally given by a satisfaction relation $\models$ between program states and CTL formulas. In the following, we write $s \models \phi$ if and only if the formula ϕ holds in the program state $s \in \Sigma$. We assume that the satisfaction relation for atomic propositions is given. The satisfaction relation for other CTL formulas is formally defined as follows:

$$\begin{array}{lcl} s \models \neg\phi & \Leftrightarrow & \neg(s \models \phi) \\ s \models \phi_1 \wedge \phi_2 & \Leftrightarrow & s \models \phi_1 \wedge s \models \phi_2 \\ s \models \phi_1 \vee \phi_2 & \Leftrightarrow & s \models \phi_1 \vee s \models \phi_2 \\ s \models \mathsf{A}\varphi & \Leftrightarrow & \forall \sigma \in T(s) \colon \sigma \models \varphi \\ s \models \mathsf{E}\varphi & \Leftrightarrow & \exists \sigma \in T(s) \colon \sigma \models \varphi \end{array} \tag{3.1}$$

where $T(s) \in \mathcal{P}(\Sigma^{+\infty})$ denotes the set of all program traces starting in the state $s \in \Sigma$. The semantics of trace formulas φ is defined below:

$$\begin{array}{lcl} \sigma \models \mathsf{X}\phi & \Leftrightarrow & \sigma[1] \models \phi \\ \sigma \models \mathsf{G}\phi & \Leftrightarrow & \forall 0 \leq i \colon \sigma[i] \models \phi \\ \sigma \models \phi_1\ \mathsf{U}\ \phi_2 & \Leftrightarrow & \exists 0 \leq i \colon \sigma[i] \models \phi_2 \wedge \forall 0 \leq j < i \colon \sigma[j] \models \phi_1 \end{array} \tag{3.2}$$

where $\sigma[i]$ denotes the program state at position i on the trace $\sigma \in \Sigma^{+\infty}$. We refer to [1] for further details.

4 Program Semantics for CTL Properties

In the following, we derive a program semantics that is *sound and complete* for proving a CTL property. We define the semantics inductively on the structure of a CTL formula. More specifically, for each formula ϕ, we define the *CTL abstraction* $\alpha_\phi \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ which extracts a partial function $f \colon \Sigma \rightharpoonup \mathbb{O}$ from program states to ordinals from a given set of sequences $T \in \mathcal{P}(\Sigma^{+\infty})$ by building upon the CTL abstractions of the sub-formulas of ϕ. The domain of f coincides with the set of program states that satisfy ϕ. Ordinal values are needed to support programs with possibly unbounded non-determinism [18]. The definition of α_ϕ for each CTL formula is summarized in Fig. 2 and explained in more detail below. We use the CTL abstraction to define the program semantics $\Lambda_\phi \colon \Sigma \rightharpoonup \mathbb{O}$ for a formula ϕ by abstraction of the maximal trace semantics Λ.

Definition 1. *Given a CTL formula ϕ and the corresponding CTL abstraction $\alpha_\phi \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$, the program semantics $\Lambda_\phi \colon \Sigma \rightharpoonup \mathbb{O}$ for ϕ is defined as $\Lambda_\phi \stackrel{def}{=} \alpha_\phi(\Lambda)$, where Λ is the maximal trace semantics (cf. Eq. 2.1).*

ϕ	$\alpha_\phi \colon \mathcal{P}\left(\Sigma^{+\infty}\right) \to (\Sigma \rightharpoonup \mathbb{O})$
a	$\alpha_a(T) \stackrel{\text{def}}{=} \lambda s \in \mathrm{st}(T).\ \begin{cases} 0 & s \models a \\ \text{undefined} & \text{otherwise} \end{cases}$
$\mathsf{Q}\mathsf{X}\phi$	$\alpha_{\mathsf{Q}\mathsf{X}\phi}(T) \stackrel{\text{def}}{=} \lambda s \in \mathrm{st}(T).\ \begin{cases} 0 & s \in \mathrm{trans}_{\mathsf{Q}}(\mathrm{dom}(\alpha_\phi(T))) \\ \text{undefined} & \text{otherwise} \end{cases}$
$\mathsf{Q}(\phi_1\ \mathsf{U}\ \phi_2)$	$\alpha_{\mathsf{Q}(\phi_1\mathsf{U}\phi_2)}(T) \stackrel{\text{def}}{=} \alpha^{\mathrm{rk}}_{\mathsf{Q}}(\alpha^{\mathrm{sq}}_{\mathsf{Q}(\phi_1\mathsf{U}\phi_2)}(T))$
$\mathsf{Q}\mathsf{G}\phi$	$\alpha_{\mathsf{Q}\mathsf{G}\phi}(T) \stackrel{\text{def}}{=} \mathrm{gfp}^{\sqsubseteq}_{\alpha_\phi(T)}\ \Theta_{\mathsf{Q}\mathsf{G}\phi}$ $\Theta_{\mathsf{Q}\mathsf{G}\phi}(f) \stackrel{\text{def}}{=} \lambda s.\ \begin{cases} f(s) & s \in \mathrm{dom}(f) \cap \mathrm{trans}_{\mathsf{Q}}(\mathrm{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases}$
$\neg\phi$	$\alpha_{\neg\phi}(T) \stackrel{\text{def}}{=} \lambda s \in \mathrm{st}(T).\ \begin{cases} 0 & s \notin \mathrm{dom}(\alpha_\phi(T)) \\ \text{undefined} & \text{otherwise} \end{cases}$
$\phi_1 \wedge \phi_2$	$\alpha_{\phi_1\wedge\phi_2}(T) \stackrel{\text{def}}{=} \lambda s \in \mathrm{st}(T).\ \begin{cases} \sup\{f_1(s), f_2(s)\} & s \in \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) \\ \text{undefined} & \text{otherwise} \end{cases}$
$\phi_1 \vee \phi_2$	$\alpha_{\phi_1\vee\phi_2}(T) \stackrel{\text{def}}{=} \lambda s \in \mathrm{st}(T).\ \begin{cases} \sup\{f_1(s), f_2(s)\} & s \in \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) \\ f_1(s) & s \in \mathrm{dom}(f_1) \setminus \mathrm{dom}(f_2) \\ f_2(s) & s \in \mathrm{dom}(f_2) \setminus \mathrm{dom}(f_1) \\ \text{undefined} & \text{otherwise} \end{cases}$

Fig. 2. CTL abstraction $\alpha_\phi \colon \mathcal{P}\left(\Sigma^{+\infty}\right) \to (\Sigma \rightharpoonup \mathbb{O})$ for each CTL formula ϕ. The function $\mathrm{trans}_{\mathsf{Q}}$ stands for pre, if Q is E, or $\widetilde{\mathrm{pre}}$, if Q is A (cf. Sect. 2). The *state function* $\mathrm{st} \colon \mathcal{P}\left(\Sigma^{+\infty}\right) \to \mathcal{P}\left(\Sigma\right)$ collects all states of a given set of sequences T: $\mathrm{st}(T) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \exists \sigma' \in \Sigma^*, \sigma'' \in \Sigma^{*\infty} : \sigma' s \sigma'' \in T\}$. The *ranking abstraction* $\alpha^{\mathrm{rk}}_{\mathsf{Q}} \colon \mathcal{P}\left(\Sigma^{+}\right) \to (\Sigma \rightharpoonup \mathbb{O})$ is defined in Eq. 4.1, while the *subsequence abstraction* $\alpha^{\mathrm{sq}}_{\mathsf{Q}\mathsf{F}\phi} \colon \mathcal{P}\left(\Sigma^{+\infty}\right) \to \mathcal{P}\left(\Sigma^{+}\right)$ is defined in Eqs. 4.2 and 4.3. In the last two rows, $f_1 \stackrel{\text{def}}{=} \alpha_{\phi_1}(T)$ and $f_2 \stackrel{\text{def}}{=} \alpha_{\phi_2}(T)$.

Remarks. It may seem unintuitive to define Λ_ϕ starting from program traces rather than program states (as in Sect. 3). The reason behind this deliberate choice is that it allows placing Λ_ϕ in the hierarchy of semantics defined by Cousot [14], which is a uniform framework that makes program semantics easily comparable and facilitates explaining the similarities and correspondences between semantic models. Specifically, this enables the comparison with existing semantics for termination [18] and other liveness properties [41] (cf. Sect. 7).

It may also seem unnecessary to define Λ_ϕ to be a function. However, this choice yields a uniform treatment of CTL formulas independently of whether they express safety or liveness properties (or a combination of these). Additionally, it allows leveraging existing abstract domains [38,39] (cf. Sect. 5) to obtain a sound static analysis for CTL properties. In particular, the proof of the soundness of the static analysis (cf. Theorem 2 and [38] for more details) requires reasoning both about the domain of Λ_ϕ as well as its value.

Atomic Propositions. For an atomic proposition a, the CTL abstraction $\alpha_a \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ simply extracts from a given set T of sequences a partial function that maps the states of the sequences in T (i.e., $s \in \mathrm{st}(T)$) that satisfy a (i.e., $s \models a$) to the constant value zero, meaning that no program execution steps are needed until a is satisfied for all executions starting in those states. Thus, the domain of the corresponding program semantics $\Lambda_a \colon \Sigma \rightharpoonup \mathbb{O}$ is (cf. Definition 1) is the set of program states that satisfy a (since $\mathrm{st}(\Lambda) = \Sigma$).

Next-Formulas. Next-formulas $\mathsf{QX}\phi$ express that the next state of all traces (if Q is A) or at least one trace (if Q is E) satisfies ϕ.

The CTL abstraction $\alpha_{\mathsf{QX}\phi} \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ for a next-formula $\mathsf{QX}\phi$ (cf. Fig. 2) maps a set T of sequences to a partial function defined over the states of the sequences in T (i.e., $s \in \mathrm{st}(T)$) that are the *predecessors* of the states that satisfy ϕ, that is, the predecessors of the states in the domain of the CTL abstraction for ϕ (i.e., $s \in \mathrm{trans}_{\mathsf{Q}}(\mathrm{dom}(\alpha_\phi(T)))$). The function has constant value zero over its domain, again meaning that no program execution steps are needed until $\mathsf{QX}\phi$ is satisfied for all executions starting in those states.

Thus, the domain of the program semantics $\Lambda_{\mathsf{QX}\phi} \colon \Sigma \rightharpoonup \mathbb{O}$ is the set of states inevitably (for $\Lambda_{\mathsf{AX}\phi}$) or possibly (for $\Lambda_{\mathsf{EX}\phi}$) leading to a state in the domain $\mathrm{dom}(\Lambda_\phi)$ of the program semantics of the sub-formula ϕ (cf. Definition 1).

Until-Formulas. Until-formulas $\mathsf{Q}(\phi_1 \ \mathsf{U} \ \phi_2)$ express that some desired state (i.e., a state satisfying the sub-formula ϕ_2) is eventually reached during program execution, either in all traces (if Q is A) or in at least one trace (if Q is E), and the sub-formula ϕ_1 is satisfied in all program states encountered until then. Thus, we can observe that an until-formula is satisfied by *finite* subsequences of possibly *infinite* program traces. To reason about subsequences, we define the *subsequence function* $\mathrm{sq} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ which extracts all finite subsequences of a given set of sequences T: $\mathrm{sq}(T) \stackrel{\text{def}}{=} \{\sigma \in \Sigma^{+} \mid \exists \sigma' \in \Sigma^{*}, \sigma'' \in \Sigma^{*\infty} : \sigma'\sigma\sigma'' \in T\}$. In the following, given a formula $\mathsf{Q}(\phi_1 \ \mathsf{U} \ \phi_2)$, we define the corresponding *subsequence abstraction* $\alpha^{\mathrm{sq}}_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ which extracts the finite subsequences that satisfy $\mathsf{Q}(\phi_1 \ \mathsf{U} \ \phi_2)$ from of a set of sequences T. We can then use $\alpha^{\mathrm{sq}}_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}$ to define the CTL abstraction $\alpha_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)} \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ as shown in Fig. 2. The *ranking abstraction* $\alpha^{\mathrm{rk}}_{\mathsf{Q}} \colon \mathcal{P}(\Sigma^{+}) \to (\Sigma \rightharpoonup \mathbb{O})$ is:

$$\alpha^{\mathrm{rk}}_{\mathsf{Q}}(T) \stackrel{\text{def}}{=} \alpha^{\mathrm{v}}_{\mathsf{Q}}(\vec{\alpha}\,(T)) \tag{4.1}$$

where $\vec{\alpha} \colon \mathcal{P}(\Sigma^{+}) \to \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma \times \Sigma)$ extracts from a given set of non-empty finite sequences T the smallest transition system $\langle S, r\rangle$ that generates T : $\vec{\alpha}\,(T) \stackrel{\text{def}}{=} \langle \mathrm{st}(T), \{\langle s, s'\rangle \in \Sigma \times \Sigma \mid \exists \sigma \in \Sigma^{*}, \sigma' \in \Sigma^{*\infty} : \sigma s s' \sigma' \in T\}\rangle$ and the function $\alpha^{\mathrm{v}}_{\mathsf{Q}} \colon \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma \times \Sigma) \to (\Sigma \rightharpoonup \mathbb{O})$ provides the rank of the elements

in the domain of the transition relation of the transition system:

$$\alpha^{\mathrm{v}}_{\mathsf{Q}}\langle S, r\rangle s \stackrel{\text{def}}{=} \begin{cases} 0 & \forall s' \in S : \langle s, s'\rangle \notin r \\ \mathrm{bnd}_{\mathsf{Q}} \left\{ \alpha^{\mathrm{v}}_{\mathsf{Q}}\langle S, r\rangle s' + 1 \middle| \begin{array}{c} s \neq s', \langle s, s'\rangle \in r, \\ s' \in \mathrm{dom}(\alpha^{\mathrm{v}}_{\mathsf{Q}}\langle S, r\rangle) \end{array} \right\} & \text{otherwise} \end{cases}$$

where $\mathrm{bnd}_{\mathsf{Q}}$ stands for sup, if Q is A, or inf, if Q is E. The CTL abstraction $\alpha_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}$ (resp. $\alpha_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}$) maps the states $\mathrm{st}(T)$ of a given set of sequences T that satisfy $\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)$ to an upper bound (resp. lower bound) on the number of program execution steps until the sub-formula ϕ_2 is satisfied, for all (resp. at least one of the) executions starting in those states.

Existential Until-Formulas. The subsequence abstraction $\alpha^{\mathrm{sq}}_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}$ for a formula $\mathsf{E}(\phi_1 \mathsf{U} \phi_2)$ extracts from a given a set of sequences T the finite subsequence of states that terminate in a state satisfying ϕ_2 and all predecessor states satisfy ϕ_1 (and not ϕ_2). It is defined as follows:

$$\begin{aligned} \alpha^{\mathrm{sq}}_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}(T) &\stackrel{\text{def}}{=} \overline{\alpha}_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}[\mathrm{dom}(\alpha_{\phi_1}(T))][\mathrm{dom}(\alpha_{\phi_2}(T))]T \\ \overline{\alpha}_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}[S_1][S_2]T &\stackrel{\text{def}}{=} \{\sigma s \in \mathrm{sq}(T) \mid \sigma \in (S_1 \setminus S_2)^*, s \in S_2\} \end{aligned} \tag{4.2}$$

where S_1 is the set of states that satisfy the sub-formula ϕ_1 (i.e., $\mathrm{dom}(\alpha_{\phi_1}(T))$), and S_2 is the set of desired states (i.e., $\mathrm{dom}(\alpha_{\phi_2}(T))$).

Universal Until-Formulas. A finite subsequence of states satisfies a universal until-formula $\mathsf{A}(\phi_1 \mathsf{U} \phi_2)$ if and only if it terminates in a state satisfying ϕ_2, all predecessor states satisfy ϕ_1, *and all other sequences with a common prefix also terminate in a state satisfying* ϕ_2 *(and all its predecessors satisfy* ϕ_1*)*, i.e., the program reaches a desired state (via states that satisfy ϕ_1) independently of the non-deterministic choices made during execution. We define the *neighborhood* of a sequence of states σ in a given set T as the set of sequences $\sigma' \in T$ with a common prefix with σ: $\mathrm{nbhd}(\sigma, T) \stackrel{\text{def}}{=} \{\sigma' \in T \mid \mathrm{pf}(\sigma) \cap \mathrm{pf}(\sigma') \neq \emptyset\}$, where the *prefixes function* $\mathrm{pf}\colon \Sigma^{+\infty} \to \mathcal{P}(\Sigma^{+\infty})$ yields the set of non-empty prefixes of a sequence $\sigma \in \Sigma^{+\infty}$: $\mathrm{pf}(\sigma) \stackrel{\text{def}}{=} \{\sigma' \in \Sigma^{+\infty} \mid \exists \sigma'' \in \Sigma^{*\infty} : \sigma = \sigma'\sigma''\}$.

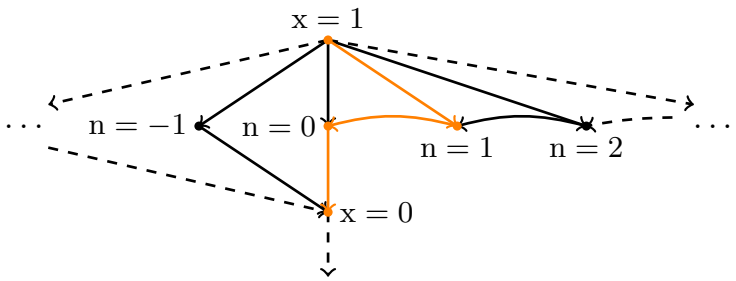
We can now defined the subsequence abstraction $\alpha^{\mathrm{sq}}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}$:

$$\begin{aligned} \alpha^{\mathrm{sq}}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}(T) &\stackrel{\text{def}}{=} \overline{\alpha}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}[\mathrm{dom}(\alpha_{\phi_1}(T))][\mathrm{dom}(\alpha_{\phi_2}(T))]T \\ \overline{\alpha}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}[S_1][S_2]T &\stackrel{\text{def}}{=} \left\{ \sigma s \in \mathrm{sq}(T) \middle| \begin{array}{c} \sigma \in (S_1 \setminus S_2)^*, s \in S_2, \\ \mathrm{nbhd}(\sigma, \mathrm{sf}(T) \cap \overline{S_2}^{+\infty}) = \emptyset, \\ \mathrm{nbhd}(\sigma, \mathrm{sf}(T) \cap Z) = \emptyset \end{array} \right\} \end{aligned} \tag{4.3}$$

where the *suffixes function* $\mathrm{sf}\colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ yields the set of non-empty suffixes of a set of sequences T: $\mathrm{sf}(T) \stackrel{\text{def}}{=} \bigcup \{\sigma \in \Sigma^{+\infty} \mid \exists \sigma' \in \Sigma^* : \sigma'\sigma \in T\}$, and $Z \stackrel{\text{def}}{=} \left\{\sigma s \sigma' \in \Sigma^{+\infty} \mid \sigma \in \Sigma^* \wedge s \in \overline{S_1 \cup S_2} \wedge \sigma' \in \Sigma^{+\infty}\right\}$ is the set of sequences of states in which at least one state satisfies neither ϕ_1 nor ϕ_2. The last two

conjuncts in the definition of the helper function $\overline{\alpha}_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}[S_1][S_2]$ ensure that a finite subsequence satisfies $\mathsf{A}(\phi_1\ \mathsf{U}\ \phi_2)$ only if it does not have a common prefix with any subsequence of T that never reaches a desired state in S_2 (i.e., $\mathrm{nbhd}(\sigma, \mathrm{sf}(T) \cap \overline{S_2}^{+\infty}) = \emptyset$) and with any subsequence that contains a state that does not belong to S_1 and S_2 (i.e., $\mathrm{nbhd}(\sigma, \mathrm{sf}(T) \cap Z) = \emptyset$).

Example 1. Let us consider again the acquire/release program of Fig. 1 and let T be the set of its traces. The suffixes starting at program point 2 of the traces in T can be visualized as follows:



Observe that these sequences form a neighborhood in the set $\mathrm{sf}(T)$ of suffixes of T (i.e., the set of all these sequences is the neighborhood $\mathrm{nbhd}(\sigma, \mathrm{sf}(T))$ of any sequence σ in the set). In the following, we write x_i and n_i for the states denoted above by $\mathrm{x} = i$ and $\mathrm{n} = i$, respectively.

Let us consider the universal until-formula $\mathsf{A}(\mathrm{x} = 1\ \mathsf{U}\ \mathrm{x} = 0)$. The set of desired states is $S_2 = \{\mathrm{x}_0\}$ and $S_1 = \{\mathrm{x}_1\} \cup \{\mathrm{n}_i \mid i \in \mathbb{Z}\}$ is the set of states that satisfy $\mathrm{x} = 1$. All sequences in the neighborhood have prefixes of the form σs where $\sigma = \mathrm{x}_1 \cdots \in (S_1 \cap \overline{S_2})^*$ and $s = \mathrm{x}_0 \in S_2$. Thus, the neighborhood of any subsequence σs does not contain sequences in $\overline{S_2}^{+\infty}$ that never reach the desired state x_0 (i.e., $\mathrm{nbhd}(\sigma s, \mathrm{sf}(T) \cap \overline{S_2}^{+\infty}) = \emptyset$). Furthermore, the neighborhood does not contain sequences in Z in which at least one state neither satisfies $\mathrm{x} = 1$ nor $\mathrm{x} = 0$ (i.e., $\mathrm{nbhd}(\sigma, \mathrm{sf}(T) \cap Z) = \emptyset$). Therefore, the until-formula $\mathsf{A}(\mathrm{x} = 1\ \mathsf{U}\ \mathrm{x} = 0)$ is satisfied at program point 2.

Let us consider now the formula $\mathsf{A}(\mathrm{x} = 1 \wedge 0 \leq \mathrm{n}\ \mathsf{U}\ \mathrm{x} = 0)$. Again, all sequences in the neighborhood eventually reach the desired state x_0. However, in this case, the set S_1 is limited to states with non-negative values for n, i.e., $S_1 = \{\mathrm{x}_1\} \cup \{\mathrm{n}_i \mid 0 \leq i\}$. Thus, the neighborhood also contains sequences in which at least one state satisfies neither $\mathrm{x} = 1 \wedge 0 \leq \mathrm{n}$ nor $\mathrm{x} = 0$ (e.g., the sequence $\mathrm{x}_1 \mathrm{n}_{-1} \ldots$). Hence $\mathsf{A}(\mathrm{x} = 1 \wedge 0 \leq \mathrm{n}\ \mathsf{U}\ \mathrm{x} = 0)$ is not satisfied at program point 2 since $\mathrm{nbhd}(\sigma, \mathrm{sf}(T) \cap Z) \neq \emptyset$. Instead, the existential until-formula $\mathsf{E}(\mathrm{x} = 1 \wedge 0 \leq \mathrm{n}\ \mathsf{U}\ \mathrm{x} = 0)$ is satisfied since, for instance, the subsequence σs where $\sigma = \mathrm{x}_1 \mathrm{n}_1$ and $s = \mathrm{x}_0$ satisfies $(\mathrm{x} = 1 \wedge 0 \leq \mathrm{n}\ \mathsf{U}\ \mathrm{x} = 0)$.

Until Program Semantics. We now have all the ingredients that define the program semantics $\Lambda_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)} : \Sigma \rightharpoonup \mathbb{O}$ for an until-formula $\mathsf{Q}(\phi_1\ \mathsf{U}\ \phi_2)$ (cf. Definition 1). Let $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$ be the partially ordered set for the computational order $f_1 \sqsubseteq f_2 \Leftrightarrow \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x)$. The program

semantics $\Lambda_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}$ can be expressed as a least fixpoint in $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$ as:

$$\Lambda_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)} = \mathrm{lfp}^{\sqsubseteq}_{\dot{\emptyset}} \Theta_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}[\mathrm{dom}(\Lambda_{\phi_1})][\mathrm{dom}(\Lambda_{\phi_2})]$$

$$\Theta_{\mathsf{Q}(\phi_1 \mathsf{U} \phi_2)}[S_1][S_2]f \stackrel{\text{def}}{=} \lambda s. \begin{cases} 0 & s \in S_2 \\ \mathrm{bnd}_{\mathsf{Q}} \{ f(s') + 1 \mid \langle s, s' \rangle \in \tau \} & s \in S_1 \wedge s \notin S_2 \wedge \\ & s \in \mathrm{trans}_{\mathsf{Q}}(\mathrm{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{4.4}$$

where $\dot{\emptyset}$ is the totally undefined function. The program semantics $\Lambda_{\mathsf{A}(\phi_1 \mathsf{U} \phi_2)}$ (resp. $\Lambda_{\mathsf{E}(\phi_1 \mathsf{U} \phi_2)}$) is a well-founded function mapping each program state in $\mathrm{dom}(\Lambda_{\phi_1})$ inevitably (resp. possibly) leading to a desirable state in $\mathrm{dom}(\Lambda_{\phi_2})$ to an ordinal, which represents an upper bound (resp. lower bound) on the number of program execution steps needed until a desirable state is reached.

Globally-Formulas. Globally-formulas $\mathsf{QG}\phi$ express that ϕ holds indefinitely in all traces (if Q is A) or at least one trace (if Q is E) starting in a state.

The definition of the CTL abstraction $\alpha_{\mathsf{QG}\phi} \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ for $\mathsf{QG}\phi$ given in Fig. 2 retains the value of the CTL abstraction corresponding to the sub-formula ϕ. Intuitively, each iteration discards the states that satisfy ϕ (i.e., the states in $\mathrm{dom}(\alpha_\phi(T))$) but branch to (sub)sequences of T that do not satisfy $\mathsf{QG}\phi$. Preserving the value of α_ϕ provides more information than just mapping each state to the constant value zero. For instance, the CTL abstraction $\alpha_{\mathsf{AGAF}\phi}$ for a globally-formula $\mathsf{AGAF}\phi$ provides an upper bound on the number of program execution steps needed until the *next occurrence* of ϕ is satisfied, for all executions starting in the corresponding state.

The corresponding program semantics $\Lambda_{\mathsf{QG}\phi} \colon \Sigma \rightharpoonup \mathbb{O}$ (cf. Definition 1) preserves the value of Λ_ϕ for each state satisfying the sub-formula ϕ and inevitably (if Q is A) or possibly (if Q is E) leading only to other states that also satisfy ϕ.

Other Formulas. We are left with describing the CTL abstraction of $\neg\phi$, $\phi \wedge \phi$, and $\phi \vee \phi$ defined in Fig. 2. For a negation $\neg\phi$, the CTL abstraction $\alpha_{\neg\phi}$ maps each program state that does not satisfy ϕ to the value zero. The CTL abstraction for a binary formula $\phi_1 \wedge \phi_2$ or $\phi_1 \vee \phi_2$ retains the largest value of the functions Λ_{ϕ_1} and Λ_{ϕ_2} for each program state satisfying both ϕ_1 and ϕ_2; for a disjunction $\phi_1 \vee \phi_2$, it also retains the value of the function for each program state satisfying either sub-formula.

Theorem 1. *A program satisfies a CTL formula ϕ for all traces starting from a given set of states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\Lambda_\phi)$.*

Proof. The proof proceeds by induction over the structure of the CTL formula ϕ. The base case are atomic propositions a for which the proof is immediate.

For a next-formulas $\mathsf{QX}\phi$, by induction hypothesis, $\mathrm{dom}(\Lambda_\phi)$ coincides with the set of states that satisfy ϕ. By Definition 1 and the definition of $\alpha_{\mathsf{QX}\phi}$ in Fig. 2,

the domain of $\Lambda_{\mathsf{QX}\phi}$ coincides with $\mathrm{trans}_{\mathsf{Q}}(\mathrm{dom}(\alpha_\phi(T)))$. Thus, by definition of $\mathrm{trans}_{\mathsf{Q}}$, we have that $\mathrm{dom}(\Lambda_{\mathsf{QX}\phi})$ coincides with the set of states that satisfy $\mathsf{QX}\phi$ (cf. Eqs. 3.1 and 3.2).

For an until-formula $\mathsf{Q}(\phi_1\ \mathsf{U}\ \phi_2)$, by induction hypothesis, $\mathrm{dom}(\Lambda_{\phi_1})$ and $\mathrm{dom}(\Lambda_{\phi_2})$ coincide with the set of states that satisfy ϕ_1 and ϕ_2, respectively. We have $\Lambda_{\mathsf{Q}}(\phi_1\ \mathsf{U}\ \phi_2) = \Theta_{\mathsf{Q}(\phi_1\mathsf{U}\phi_2)}[\mathrm{dom}(\Lambda_{\phi_1})][\mathrm{dom}(\Lambda_{\phi_2})](\Lambda_{\mathsf{Q}}(\phi_1\ \mathsf{U}\ \phi_2))$ from Eq. 4.4. Therefore, by definition of $\Theta_{\mathsf{Q}(\phi_1\mathsf{U}\phi_2)}$, $\mathrm{dom}(\Lambda_{\mathsf{Q}}(\phi_1\ \mathsf{U}\ \phi_2))$ coincides with the states that satisfy ϕ_2 and all states that satisfy ϕ_1 and inevitably (if Q is A) or possibly (if Q is E) lead to states that satisfy ϕ_2. So $\mathrm{dom}(\Lambda_{\mathsf{Q}}(\phi_1\ \mathsf{U}\ \phi_2))$ coincides with the states that satisfy $\mathsf{Q}(\phi_1\ \mathsf{U}\ \phi_2)$ (cf. Eqs. 3.1 and 3.2).

For a globally-formula $\mathsf{QG}\phi$, by induction hypothesis, $\mathrm{dom}(\Lambda_\phi)$ coincides with the set of states that satisfy ϕ. By Definition 1 and the definition of $\alpha_{\mathsf{QG}\phi}$ in Fig. 2, we have that $\Lambda_{\mathsf{QG}\phi} = \Theta_{\mathsf{QG}\phi}(\Lambda_{\mathsf{QG}\phi})$. Therefore, by definition of $\Theta_{\mathsf{QG}\phi}$, we have that $\mathrm{dom}(\Lambda_{\mathsf{QG}\phi})$ coincides with the states that satisfy ϕ inevitably (if Q is A) or possibly (if Q is E) lead to other states that satisfy ϕ. So $\mathrm{dom}(\Lambda_{\mathsf{QG}\phi})$ coincides with the states that satisfy $\mathsf{QG}\phi$ (cf. Eqs. 3.1 and 3.2).

Finally, all other cases ($\neg\phi$, $\phi_1 \wedge \phi_2$, and $\phi_1 \vee \phi_2$) follow immediately from the induction hypothesis, the semantics of the CTL formulas (cf. Eq. 3.1) and the definition of the corresponding program semantics (cf. Definition 1 and the CTL abstractions in Fig. 2). □

The program semantics for a CTL formula is not computable when the program state space is infinite. In the next section, we present decidable abstractions by means of piecewise-defined functions [38,39].

5 Static Analysis for CTL Properties

We recall here the features of the abstract domain of piecewise-defined functions [39] that are relevant for our purposes, and describe the new elements that we need to introduce to obtain a static analysis for proving CTL properties. We refer to [38] for an exhaustive presentation of the original abstract domain.

For illustration, we model a program using a control flow graph $\langle \mathcal{L}, E \rangle$, where $\mathcal{L}$ is the set of program points and $E \subseteq \mathcal{L} \times A \times \mathcal{L}$ is the set of edges in the control flow graph. Each edge is labeled by an action $s \in A$; possible actions are skip, a boolean condition b, or an assignment $x := e$. In the following, we write $out(l)$ to denote the set of outgoing edges from a program point l.

Piecewise-Defined Functions Abstract Domain. An element $t \in \mathcal{T}$ of the abstract domain is a piecewise-defined partial function represented by a *decision tree*, where the decision nodes are labeled by linear constraints over the program variables, and the leaf nodes are labeled by functions of the program variables. The decision nodes recursively partition the space of possible values of the program variables, and the leaf nodes represent the value of the function corresponding to each partition. An example of (simplified) decision tree representation of a piecewise-defined function is shown in Fig. 3.

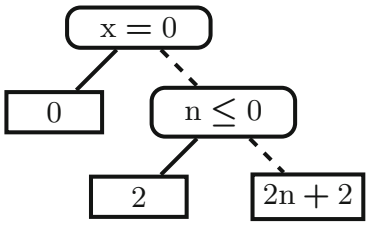


Fig. 3. Simplified decision tree representation of the piecewise-defined function inferred at program point 4 of the program of Fig. 1 (cf. Eq. 1.1). Each constraint is satisfied by the left subtree of the decision node, while the right subtree satisfies its negation. The leaves represent partial functions whose domain is determined by the constraints satisfied along the path to the leaf.

Specifically, the decision nodes are labeled by linear constraints supported by an existing underlying numerical domain, i.e., interval [15] constraints (of the form $\pm \mathrm{x} \leq \mathrm{c}$), octagonal [30] constraints (of the form $\pm \mathrm{x}_i \pm \mathrm{x}_j \leq c$), or polyhedral [19] constraints (of the form $\mathrm{c}_1 \cdot \mathrm{x}_1 + \cdots + \mathrm{c}_k \cdot \mathrm{x}_k \leq \mathrm{c}_{k+1}$). The leaf nodes are labeled by *affine functions* of the program variables (of the form $\mathrm{m}_1 \cdot \mathrm{x}_1 + \cdots + \mathrm{m}_k \cdot \mathrm{x}_k + \mathrm{q}$), or the special elements $\bot$ and $\top$, which explicitly represent undefined functions. The element $\top$ is introduced to manifest an irrecoverable precision loss of the analysis. We also support *lexicographic* affine *functions* $(\mathrm{f}_k, \ldots, \mathrm{f}_1, \mathrm{f}_0)$ in the isomorphic form of ordinals $\omega^k \cdot \mathrm{f}_k + \cdots + \omega \cdot \mathrm{f}_1 + \mathrm{f}_0$ [29,40].

The partitioning is dynamic: during the analysis of a control flow graph, partitions (i.e. decision nodes and constraints) are modified by assignments and split (i.e., added) by boolean conditions and when merging control flows. More specifically, for each action $s \in A$, we define sound over-approximating abstract transformers $[\![s]\!]_o : \mathcal{T} \to \mathcal{T}$ as well as *new under-approximating abstract transformers* $[\![s]\!]_u : \mathcal{T} \to \mathcal{T}$. These transformers always increase by one the value of the functions labeling the leaves of a given decision tree to count the number of executed program steps (i.e., actions in the control flow graph). The transformers for boolean conditions and assignments additionally modify the decision nodes by building upon the underlying numerical abstract domain. For instance, the abstract transformer $[\![b]\!]_o$ (resp. $[\![b]\!]_u$) for a boolean condition b uses the underlying numerical domain to obtain an over-approximation (resp. an under-approximation) of b as a set of linear constraints; then it adds these constraints to the given decision tree and discards the paths that become unfeasible (because they do not satisfy the added constraints). Let $\{n \leq 0\}$ (resp. $\{n = 0\}$) be the set of constraints obtained by $[\![b]\!]_o$ (resp. $[\![b]\!]_u$) for the boolean condition $b \equiv n \leq 0 \wedge n\%2 = 0$; then, given the right subtree in Fig. 3, $[\![b]\!]_o$ (resp. $[\![b]\!]_u$) would discard the path leading to the leaf with value $2n+2$ by replacing it with a leaf with undefined value $\bot$ (resp. replace $n \leq 0$ with $n = 0$ and replace $2n+2$ with $\bot$). Decision trees are merged using either the approximation join $\curlyvee$ or the computational join $\sqcup$. Both join operators add missing decision nodes from either of the given trees; $\curlyvee$ retains the leaves that are labeled with an undefined function in at least one of the given trees, while $\sqcup$ preserves the leaves that are labeled with a defined function over the leaves labeled with $\bot$ (but preserves the

leaves labeled with $\top$ over all other leaves). To minimize the cost of the analysis and to enforce termination, a (dual) widening operator limits the height of the decision trees and the number of maintained partitions.

$$\Lambda^\sharp_a \stackrel{\text{def}}{=} \lambda l.\ \textsc{reset}\ [\![a]\!]\ (\bot) \tag{5.1}$$

$$\Lambda^\sharp_{\mathsf{Q}\mathsf{X}\phi} \stackrel{\text{def}}{=} \lambda l.\ \textsc{zero}\left(\bigsqcup_{(l,s,l')\in out(l)}^{\mathsf{Q}} [\![s]\!]_{\mathsf{Q}}\ (\Lambda^\sharp_\phi(l'))\right) \tag{5.2}$$

$$\Lambda^\sharp_{\mathsf{Q}(\phi_1\mathsf{U}\phi_2)} \stackrel{\text{def}}{=} \mathrm{lfp}^\sharp_{\lambda l.\bot}\ \lambda m.\ \lambda l.\ \textsc{until}\left[\!\!\left[\Lambda^\sharp_{\phi_1}(l), \Lambda^\sharp_{\phi_2}(l)\right]\!\!\right]\left(\bigsqcup_{(l,s,l')\in out(l)}^{\mathsf{Q}} [\![s]\!]_{\mathsf{Q}}\ (m(l'))\right) \tag{5.3}$$

$$\Lambda^\sharp_{\mathsf{Q}\mathsf{G}\phi} \stackrel{\text{def}}{=} \mathrm{gfp}^\sharp_{\Lambda^\sharp_\phi}\ \lambda m.\ \lambda l.\ \textsc{mask}\left[\!\!\left[\bigsqcup_{(l,s,l')\in out(l)}^{\mathsf{Q}} [\![s]\!]_{\mathsf{Q}}\ (m(l'))\right]\!\!\right](m(l)) \tag{5.4}$$

$$\Lambda^\sharp_{\neg\phi} \stackrel{\text{def}}{=} \lambda l.\ \textsc{complement}(\Lambda^\sharp_\phi(l)) \tag{5.5}$$

$$\Lambda^\sharp_{\phi_1\wedge\phi_2} \stackrel{\text{def}}{=} \lambda l.\ \Lambda^\sharp_{\phi_1}(l) \curlyvee \Lambda^\sharp_{\phi_2}(l) \tag{5.6}$$

$$\Lambda^\sharp_{\phi_1\vee\phi_2} \stackrel{\text{def}}{=} \lambda l.\ \Lambda^\sharp_{\phi_1}(l) \sqcup \Lambda^\sharp_{\phi_2}(l) \tag{5.7}$$

Fig. 4. Abstract program semantics $\Lambda^\sharp_\phi$ for each CTL formula ϕ. The join operator $\bigsqcup^{\mathsf{Q}}$ and the abstract transformer $[\![s]\!]_{\mathsf{Q}}$ respectively stand for $\sqcup$ and $[\![s]\!]_u$, if Q is E, or $\curlyvee$ and $[\![s]\!]_o$, if Q is A. With abuse of notation, we use $\bot$ to denote a decision tree with a single undefined leaf node.

Abstract Program Semantics for CTL Properties. The abstract program semantics $\Lambda^\sharp_\phi : \mathcal{L} \to \mathcal{T}$ for a CTL formula ϕ maps each program point $l \in \mathcal{L}$ to an element $t \in \mathcal{T}$ of the piecewise-defined functions abstract domain. The definition of $\Lambda^\sharp_\phi$ for each CTL formula ϕ is summarized in Fig. 4 and explained in some detail below. More details and formal definitions can be found in [37,38].

The analysis starts with the totally undefined function (i.e., a decision tree that consists of a single leaf with undefined value $\bot$) at the final program points (i.e., nodes without outgoing edges in the control flow graph). Then it proceeds backwards through the control flow graph, taking the encountered actions into account, and joining decision trees when merging control flows. For existential CTL properties, the analysis uses the under-approximating abstract transformers $[\![s]\!]_u$ for each action s, to ensure that program states that do not satisfy the CTL property are discarded (i.e., removed from the domain of the current piecewise-defined function), and joins decision trees using the computational join $\sqcup$, to ensure that the current piecewise-defined function remains defined over states that satisfy the CTL property in at least one of the merged control flows. Dually, for universal CTL properties, the analysis uses the over-approximating abstract transformers $[\![s]\!]_o$ and joins decision trees using the approximation join $\curlyvee$, to ensure that the current piecewise-defined function remains defined only over states that satisfy the CTL property in all of the merged control flows.

At each program point, the analysis additionally performs operations that are specific to the considered CTL formula ϕ. For an atomic proposition a (cf. Eq. 5.1), the analysis performs a RESET $[\![a]\!]$ operation, which is analogous to the under-approximating transformer for boolean conditions but additionally replaces all the leaves that satisfy a with leaves labeled with the function with value zero. For example, given the atomic proposition $n = 0$ and the right subtree in Fig. 3, RESET $[\![n = 0]\!]$ would replace the constraint $n \leq 0$ with $n = 0$, the leaf $2n + 2$ with $\bot$ and the leaf 2 with 0. For a next-formula $\mathsf{QX}\phi$ (cf. Eq. 5.2), the analysis approximates the effect of the transition from each program point l to each successor program point l' and performs a ZERO operation to replace all defined functions labeling the leaves of the so obtained decision tree with the function with value zero. For an until-formula $\mathsf{Q}(\phi_1 \ \mathsf{U} \ \phi_2)$ (cf. Eq. 5.4), the analysis performs an ascending iteration with widening [13]. At each iteration, the analysis approximates the effect of the transition from each program point l to each successor program point l' and performs an UNTIL operation to model the until temporal operator: UNTIL replaces with the function with value zero all leaves that correspond to defined leaves in the decision tree $\Lambda^\natural_{\phi_2}(l)$ obtained for ϕ_2, and retains all leaves that are labeled with an undefined function in both $\Lambda^\natural_{\phi_1}(l)$ and $\Lambda^\natural_{\phi_1}(l)$. For a globally-formula $\mathsf{QG}\phi$ (cf. Eq. 5.5), the analysis performs a descending iteration with dual widening [41], starting from the abstract semantics $\Lambda^\natural_\phi$ obtained for ϕ. At each iteration, the MASK operation models the globally temporal operator: it discards all defined partitions in $\Lambda^\natural_\phi(l)$ that become undefined as a result of the transition from each program point l to each successor program point l'; at the limit, the only defined partitions are those that remain defined across transitions and thus satisfy the globally-formula. For a negation formula $\neg\phi$ (cf. Eq. 5.5), the analysis performs a COMPLEMENT operation on the decision tree $\Lambda^\natural_\phi(l)$ obtained for ϕ at each program point l; COMPLEMENT replaces all defined functions labeling the leaves of a decision tree with $\bot$, and all $\bot$ with the function with value zero. Note that $\Lambda^\natural_\phi$ is an abstraction of Λ_ϕ and thus not all undefined partitions in $\Lambda^\natural_\phi$ necessarily correspond to undefined partitions in Λ_ϕ. Leaves that are undefined in $\Lambda^\natural_\phi$ due to this uncertainty are labeled with $\top$, and are left unchanged by COMPLEMENT to guarantee the soundness of the analysis. Finally, for binary formulas $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$, the abstract semantics $\Lambda^\natural_{\phi_1 \wedge \phi_2}$ and $\Lambda^\natural_{\phi_1 \vee \phi_2}$ (cf. Eqs. 5.6 and 5.7) merge the decision trees obtained for ϕ_1 and ϕ_2 using the approximation join $\curlyvee$ and the computational join $\sqcup$, respectively.

The abstract program semantics $\Lambda^\natural_\phi$ for each CTL formula ϕ is *sound* with respect to the approximation order $f_1 \preccurlyeq f_2 \Leftrightarrow \mathrm{dom}(f_1) \supseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x)$, which means that the abstract semantics $\Lambda^\natural_\phi$ *over-approximates* the value of the concrete semantics Λ_ϕ and *under-approximates* its domain of definition $\mathrm{dom}(\Lambda_\phi)$. In this way, the abstraction provides sufficient preconditions for the CTL property ϕ: if the abstraction is defined for a state then that state satisfies ϕ.

Theorem 2. *A program satisfies a CTL formula ϕ for all traces starting from a given set of states $\mathcal{I}$ if $\mathcal{I} \subseteq \mathrm{dom}(\gamma(\Lambda^\natural_\phi))$.*

Proof. (Sketch). The proof proceeds by induction over the structure of the formula ϕ. The base case are atomic propositions for which the proof is immediate.

For a next-formula $\mathsf{Q}\mathsf{X}\phi$, by induction hypothesis, $\mathrm{dom}(\Lambda^\natural_\phi)$ is a subset of the set of states that satisfy ϕ. Using the over-approximating transformers $[\![s]\!]_o$ together with the approximation join $\curlyvee$ (resp. the under-approximating transformers $[\![s]\!]_u$ together with the computational join $\sqcup$) ensures that $\Lambda^\natural_{\mathsf{Q}\mathsf{X}\phi}$ soundly under-approximates the set of states that satisfy $\mathsf{Q}\mathsf{X}\phi$.

For an until-formula $\mathsf{Q}(\phi_1\ \mathsf{U}\ \phi_2)$, by induction hypothesis, $\mathrm{dom}(\Lambda^\natural_{\phi_1})$ and $\mathrm{dom}(\Lambda^\natural_{\phi_2})$ are a subset of the set of states that satisfy ϕ_1 and ϕ_2, respectively. By definition, $\Lambda_{\mathsf{Q}(\phi_1\mathsf{U}\phi_2)}$ is the limit of an ascending iteration sequence using widening. Again, using the over-approximating transformers $[\![s]\!]_o$ together with the approximation join $\curlyvee$ (resp. the under-approximating transformers $[\![s]\!]_u$ together with the computational join $\sqcup$) guarantees the soundness of the analysis with respect to each transition. The soundness of each iteration without widening is then guaranteed by the definition of the UNTIL operation. The iterations with widening are allowed to be unsound but the limit of the iterations (i.e., $\Lambda_{\mathsf{Q}(\phi_1\mathsf{U}\phi_2)}$) is guaranteed to soundly under-approximate the set of states that satisfy $(\phi_1\ \mathsf{U}\ \phi_2)$. We refer to [38] for a detailed proof for formulas of the form $(\mathit{true}\ \mathsf{U}\ \phi_2)$. The generalization to $(\phi_1\ \mathsf{U}\ \phi_2)$ is trivial.

For a globally-formula $\mathsf{Q}\mathsf{G}\phi$, $\Lambda_{\mathsf{Q}\mathsf{G}\phi}$ is the limit of a descending iteration sequence with dual widening, starting from $\Lambda^\natural_\phi$, which soundly under-approximates the set of states that satisfy ϕ. The soundness of each iteration is guaranteed by the definition of the MASK operation and the dual widening operator (see [38]).

The case of a negation $\neg\phi$ is non-trivial since, by induction hypothesis, $\mathrm{dom}(\Lambda^\natural_\phi)$ is a subset of the set of states that satisfy ϕ. Specifically, $\Lambda^\natural_\phi$ maps each program point $l \in \mathcal{L}$ to a decision tree whose leaves determine this under-approximation: leaves labeled with $\bot$ represent states that do not satisfy ϕ while leaves labeled with $\top$ represent states that may or may not satisfy ϕ. The COMPLEMENT operation performed by $\Lambda^\natural_{\neg\phi}$ only considers leaves labeled by $\bot$ and ignores (i.e., leaves unchanged) leaves labeled by $\top$. Thus, $\Lambda^\natural_{\neg\phi}$ soundly under-approximates the set of states that satisfy $\neg\phi$.

Finally, for binary formulas $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$, the proof follows immediately from the definition of the approximation join $\curlyvee$ and the computational join $\sqcup$ used in the definition of $\Lambda^\natural_{\phi_1\wedge\phi_2}$ and $\Lambda^\natural_{\phi_1\vee\phi_2}$, respectively. □

6 Implementation

The proposed static analysis method for proving CTL properties is implemented in the prototype static analyzer FUNCTION [13].

The implementation is in OCAML and consists of around 9K lines of code. The current front-end of FUNCTION accepts non-deterministic programs written

in a C-like syntax (without support for pointers, `struct` and `union` types). The only basic data type is mathematical integers. FuncTion accepts CTL properties written using a syntax similar to the one used in the rest of this paper, with atomic propositions written as C-like pure expressions. The abstract domain of piecewise-defined functions builds on the numerical abstract domains provided by the Apron library [24], and the under-approximating numerical operators provided by the Banal static analyzer [31].

The analysis is performed backwards on the control flow graph of a program with a standard worklist algorithm [32], using widening and dual widening at loop heads. Non-recursive function calls are inlined, while recursion is supported by augmenting the control flow graphs with call and return edges. The precision of the analysis can be tuned by choosing the underlying numerical abstract domain, by activating the extension to ordinal-value functions [40], and by adjusting the precision of the widening [13] and the widening delay. It is also possible to refine the analysis by considering only reachable states.

Experimental Evaluation. We evaluated our technique on a number of test cases obtained from various sources, and compared FuncTion against T2 [8] and Ultimate LTL Automizer [20] as well as E-HSF [4], and the prototype implementation from [10]. Figs. 5 and 6 show an excerpt of the results, which demonstrates the differences between FuncTion, T2 [8] and Ultimate LTL Automizer. The first set of test cases are programs that we used to test our implementation. The second and third set were collected from [25] and the web

No	Program	CTL Property	Result	Time
1.1	and_test.c	$\mathsf{AGAF}(\mathrm{n} = 1) \wedge \mathsf{AF}(\mathrm{n} = 0)$	✓	0.05s
1.2	and_test.c	$\mathsf{EGAF}(\mathrm{n} = 1)$	✓	0.05s
1.4	global_test.c	$\mathsf{AGEF}(\mathrm{x} \leq -10)$	✓	0.15s
1.7	or_test.c	$\mathsf{AFEG}(\mathrm{x} < -100) \vee \mathsf{AF}(\mathrm{x} = 20)$	✓	0.05s
1.8	may_term...	$\mathsf{EF}(\mathrm{exit} : \mathit{true})$	✗	-
1.9	until_test.c	$\mathsf{A}(\mathrm{x} \geq \mathrm{y}\ \mathsf{U}\ \mathrm{x} = \mathrm{y})$	✓	0.03s
1.11	fin_ex.c	$\mathsf{EGEF}(\mathrm{n} = 1)$	✓	0.04s
1.12	until_ex.c	$\mathsf{E}(\mathrm{x} \geq \mathrm{y}\ \mathsf{U}\ \mathrm{x} = \mathrm{y})$	✓	0.03s
2.3	win4.c	$\mathsf{AFAG}(\mathrm{WItemsNum} \geq 1)$	✓	0.15s
2.4	toylin.c	$(\mathrm{c} \leq 5 \wedge \mathrm{c} > 0) \vee \mathsf{AF}(\mathrm{resp} > 5)$	✗	-
3.9	cb5_safe.c	$\mathsf{A}(\mathrm{i} = 0\ \mathsf{U}\ (\mathsf{A}(\mathrm{i} = 1\ \mathsf{U}\ \mathsf{AG}(\mathrm{i} = 3)) \vee \mathsf{AG}(\mathrm{i} = 1)))$	✗	-
3.14	timer...	$\neg\mathsf{AG}(\mathrm{timer} = 0 \Rightarrow \mathsf{AF}(\mathrm{output} = 1))$	✗	-
3.15	togglec...	$\mathsf{AG}(\mathsf{AF}(\mathrm{t} = 1) \wedge \mathsf{AF}(\mathrm{t} = 0))$	✗	-
4.1	Bangalore...	$\mathsf{EF}(\mathrm{x} < 0)$	✗	-
4.2	Ex02...	$\mathrm{i} < 5 \Rightarrow \mathsf{AF}(\mathrm{exit} : \mathit{true})$	✓	0.04s
4.3	Ex07...	$\mathsf{AFEG}(\mathrm{i} = 0)$	✓	0.1s
4.4	java_Seq...	$\mathsf{EF}(\mathsf{AF}(\mathrm{j} \geq 21) \wedge \mathrm{i} = 100)$	✓	0.3s
4.5	Madrid...	$\mathsf{AF}(\mathrm{x} = 7 \wedge \mathsf{EFAG}(\mathrm{x} = 2))$	✓	0.02s

Fig. 5. Evaluation of FuncTion on selected test cases collected from various sources. All test cases were analyzed using polyhedral constraints for the decision nodes, and affine functions for the leaf nodes of the decision tree.

No	FuncTion	T2 [8]	Ultimate LTL Automizer [20]
1.1	✓	✗	✓
1.2	✓	✗	-
1.4	✓	✗	-
1.7	✓	✗ (error)	-
1.8	✗	✓	-
1.9	✓	✗	✓
1.11	✓	✗	-
1.12	✓	✗ (no implementation)	-
2.3	✓	✗	✓
2.4	✗	✗	✓
3.9	✗	-	✓
3.14	✗	-	✓
3.15	✗	-	✓
4.1	✗	✓	-
4.2	✓	✗ (out of memory)	✓
4.3	✓	✗	-
4.4	✓	✗ (error)	-
4.5	✓	✗	-

Fig. 6. Differences between FuncTion, T2, and Ultimate LTL Automizer.

interface of Ultimate LTL Automizer [20]. The fourth set are examples from the termination category of the 6th International Competition on Software Verification (SV-COMP 2017). The experiments were conducted on an Intel i7-6600U processor with 20 GB of RAM on Arch Linux with Linux 4.11 and OCaml 4.04.1.

FuncTion passes all test cases with the exception of 2.4, 3.9, 3.14, and 3.15, which fail due to imprecisions introduced by the widening, and 1.8 and 4.1, which fail due to an unfortunate interaction of the under-approximations needed for existential properties and non-deterministic assignments in the programs. However, note that for these test cases we still get some useful information. For instance, for 3.15, FuncTion infers that the CTL property is satisfied if $x < 0$.

In Fig. 6, the missing results for T2 are due to a missing conversion of the test cases to the T2 input format. The comparison with Ultimate LTL Automizer is limited to the test cases where the CTL property can be equivalently expressed in LTL (i.e., universal CTL properties). The results show that only FuncTion succeeds on numerous test cases (1.2, 1.4, 1.7, 1.11, 1.12, 4.3, 4.4, and 4.5). Ultimate LTL Automizer performs well on the supported test cases, but FuncTion still succeeds on most of the test cases provided by Ultimate LTL Automizer (not shown in Fig. 6, since there are no differences between the results of FuncTion and Ultimate LTL Automizer). Overall, none of the tools subsumes the others. In fact, we observe that their combination is more powerful than any of the tools alone, as it would succeed on all test cases.

Finally, FuncTion only succeeds on two of the industrial benchmarks from [10], while T2, E-HSF and [10] fare much better (see [8, Fig. 11]). The reason for the poor performance is that in these benchmarks the effect of function

calls is modeled as a non-deterministic assignment and this heavily impacts the precision of FuncTion. We are confident that we would obtain better results on the original benchmarks, where function calls are not abstracted away.

7 Related Work

In the recent past, a large body of work has been devoted to proving CTL properties of programs. The problem has been extensively studied for finite-state programs [7,26, etc.], while most of the existing approaches for infinite-state systems have limitations that restrict their applicability. For instance, they only support certain classes of programs [36], or they limit their scope to a subset of CTL [11], or to a single CTL property such as termination [27,34, etc.] or non-termination [2,5, etc.]. Our approach does not suffer from these limitations.

Some other approaches for proving CTL properties do not reliably support CTL formulas with arbitrary nesting of universal and existential path quantifiers [23], or support existential path quantifiers only indirectly by building upon recent work for proving non-termination [22], or by considering their universal dual [8]. In particular, the latter approach is problematic: since the universal dual of an existential until formula is non-trivial to define, the current implementation of T2 does not support such formulas (see Fig. 6). Other indirect approaches [4,10] perform unnecessary computations that result in slower runtimes (see [8, Fig. 12]). In comparison to all these approaches, our approach provides strictly more information in the form of a ranking function whose domain gives a precondition for a given CTL property and whose value estimates the number of program execution steps until the property is satisfied.

In [17], Cousot and Cousot define a trace-based semantics for a very general temporal language which subsumes LTL and CTL; this is subsequently abstracted to a state-based semantics. The abstraction has been later shown to be incomplete by Giacobazzi and Ranzato [21]. In contrast to the work of Cousot and Cousot, we do not define a trace-based semantics for CTL. The semantics that we propose is close to their state-based semantics in that their state-based semantics coincides with the domain of the functions that we define. Note that Theorem 1 is not in contrast with the result of Giacobazzi and Ranzato because completeness is proven with respect to the state-based semantics of CTL.

Finally, our abstract interpretation framework generalizes an existing framework [41] for proving guarantee and recurrence properties of programs [28]. Guarantee and recurrence properties are equivalently expressed in CTL as $\mathsf{A}(\mathit{true}\ \mathsf{U}\ \phi)$ and $\mathsf{AGA}(\mathit{true}\ \mathsf{U}\ \phi)$, respectively. In fact, we rediscover the guarantee and recurrence program semantics defined in [41] as instances of our framework: the guarantee semantics coincides with $\Lambda_{\mathsf{A}(\mathit{true}\mathsf{U}\phi)}$ (cf. Sect. 4) and the recurrence semantics coincides with $\Lambda_{\mathsf{AGA}(\mathit{true}\mathsf{U}\phi)}$ (cf. Sect. 4). The common insight with our work is the observation that CTL (sub)formulas are satisfied by finite subsequences (which can also be single states) of possibly infinite sequences. The program semantics for these (sub)formulas then counts the number of steps in these subsequences. Our work generalizes this idea to all CTL formulas and integrates the corresponding semantics in a uniform framework.

8 Conclusion and Future Work

In this paper, we have presented a new static analysis method for inferring preconditions for CTL properties of programs that overcomes the limitations of existing approaches. We have derived our static analysis within the framework of abstract interpretation by abstraction of the operational trace semantics of a program. Using experimental evidence, we have shown that our analysis is effective and performs well on a wide variety of benchmarks, and is able to prove CTL properties that are out of reach for state-of-the-art tools.

It remains for future work to investigate and improve the precision of the analysis in the presence of non-deterministic program assignments. We also plan to support LTL properties [20] or, more generally, CTL* properties [9]. This requires some form of trace partitioning [35] as the interpretation of LTL formulas is defined in terms of program executions instead of program states as CTL.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Bakhirkin, A., Piterman, N.: Finding recurrent sets with backward analysis and trace partitioning. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 17–35. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_2
3. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA, pp. 1–38 (2010)
4. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 869–882. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_61
5. Chen, H.-Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Proving nontermination via safety. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 156–171. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_11
6. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
8. Cook, B., Khlaaf, H., Piterman, N.: Faster temporal reasoning for infinite-state programs. In: FMCAD, pp. 75–82 (2014)
9. Cook, B., Khlaaf, H., Piterman, N.: On automation of CTL* verification for infinite-state systems. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 13–29. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_2
10. Cook, B., Koskinen, E.: Reasoning about nondeterminism in programs. In: PLDI, pp. 219–230 (2013)

11. Cook, B., Koskinen, E., Vardi, M.: Temporal property verification as a program analysis task. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 333–348. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_26
12. Cook, B., Koskinen, E., Vardi, M.Y.: Temporal property verification as a program analysis task - extended version. Formal Methods Syst. Des. **41**(1), 66–82 (2012)
13. Courant, N., Urban, C.: Precise widening operators for proving termination by abstract interpretation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 136–152. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_8
14. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoret. Comput. Sci. **277**(1–2), 47–103 (2002)
15. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Symposium on Programming, pp. 106–130 (1976)
16. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
17. Cousot, P., Cousot, R.: Temporal abstract interpretation. In: POPL, pp. 12–25 (2000)
18. Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: POPL, pp. 245–258(2012)
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96 (1978)
20. Dietsch, D., Heizmann, M., Langenfeld, V., Podelski, A.: Fairness modulo theory: a new approach to LTL software model checking. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 49–66. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_4
21. Giacobazzi, R., Ranzato, F.: Incompleteness of states w.r.t. traces in model checking. Inf. Comput. **204**(3), 376–407 (2006)
22. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.: Proving non-termination. In: POPL, pp. 147–158 (2008)
23. Gurfinkel, A., Wei, O., Chechik, M.: Yasm: a software model-checker for verification and refutation. In: CAV, pp. 170–174 (2006)
24. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
25. Koskinen, E.: Temporal verification of programs. Ph.D. thesis, University of Cambridge, November 2012
26. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000)
27. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, pp. 81–92 (2001)
28. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: PODC, pp. 377–410 (1990)
29. Manna, Z., Pnueli, A.: The Temporal Verification of Reactive Systems: Progress (1996)
30. Miné, A.: The octagon abstract domain. High. Order Symbolic Comput. **19**(1), 31–100 (2006)
31. Miné, A.: Inferring sufficient conditions with backward polyhedral under-approximations. Electron. Notes Theor. Comput. Sci. **287**, 89–100 (2012)

32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, (1999)
33. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57 (1977)
34. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41 (2004)
35. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM TOPLAS **29**(5), 26 (2007)
36. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. Theoret. Comput. Sci. **549**, 127–145 (2014)
37. Ueltschi, S.: Proving temporal properties by abstract interpretation. Master's thesis, ETH Zurich, Zurich, Switzerland (2017)
38. Urban, C.: Static Analysis by abstract interpretation of functional temporal properties of programs. Ph.D. thesis, École Normale Supérieure, Paris, France, July 2015
39. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: SAS, pp. 302–318 (2014)
40. Urban, C., Miné, A.: An abstract domain to infer ordinal-valued ranking functions. In: ESOP, pp. 412–431 (2014)
41. Urban, C., Miné, A.: Inference of ranking functions for proving temporal properties by abstract interpretation. Comput. Lang. Syst. Struct. **47**, 77–103 (2017)

[c26] Naïm Moussaoui Remil, **CU**, Antoine Miné

Automatic Detection of Vulnerable Variables for CTL Properties of Programs

In 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2024)

https://inria.hal.science/hal-04710215

EPiC Series in Computing

Volume 100, 2024, Pages 116–126

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Automatic Detection of Vulnerable Variables for CTL Properties of Programs

Naïm Moussaoui Remil[1], Caterina Urban[1], and Antoine Miné[2]

[1] Inria & ENS | PSL, Paris, France
{naim.moussaoui-remil,caterina.urban}@inria.fr
[2] Sorbonne Université, CNRS, LIP6, Paris, France
antoine.mine@lip6.fr

Abstract

We present our tool FuncTion-V for the automatic identification of the minimal sets of program variables that an attacker can control to ensure an undesirable program property. FuncTion-V supports program properties expressed in Computation Tree Logic (CTL), and builds upon an abstract interpretation-based static analysis for CTL properties that we extend with an abstraction refinement process. We showcase our tool on benchmarks collected from the literature and SV-COMP 2023.

1 Introduction

Violations of program properties pose significant risks, particularly when they can be triggered by attackers [8, 14]. This paper presents our tool FuncTion-V for the automatic identification of the *minimal* set(s) of program variables that are *vulnerable* to be controlled by an attacker to *potentially violate* a (desirable) program property, or, equivalently, the minimal variable set(s) the values of which must be controlled to *ensure* an (undesirable) program property.

Let us consider the undesirable robust reachability [8, 9] of the error location in Figure 1. Here, to make sure that the error location is reached, an attacker can either (A) control the value of x ensuring that $x > 0$ (error reachable independently of the values of y and z), or (B) control the value of y and z ensuring that $y \leq z$ (error reachable independently of the value of x). In case A, we say that x is vulnerable (while y and z are *safe*), while in case B, we say that y and z are vulnerable (while x is safe). Our tool additionally infers the sufficient conditions on the vulnerable variables that ensure the property (robust reachability of the error, in this case) independently of the values of the safe variables: $x > 0$ in case A, and $y \leq z$ in case B.

Besides robust reachability [8, 9, 18, 19], FuncTion-V more generally supports program properties expressed in Computation Tree Logic (CTL) [2] (e.g., termination [16], recurrence properties [18, 19], etc.). To improve the precision of our approach, we have equipped FuncTion-V with a conflict-driven abstraction refinement-style analysis [7].

We report on the experimentation of our tool on benchmarks from [17, 18, 4, 20, 3, 6] as well as from the termination and reachSafety categories of SV-COMP 2023.

```
1            void main (int x, int y, int z) {
2                while( y > z ){
3                    y = y - x;
4                }
5                // error
6            }
7
```

Figure 1: Running Example

2 Computation Tree Logic

CTL [2] is a branching-time logic, meaning that its model of time is a tree-like structure in which each state may have several possible futures. Formulas in CTL are formed according to the following grammar:

$$\phi ::= a \mid l : a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathrm{AX}\phi \mid \mathrm{AG}\phi \mid \mathrm{A}(\phi \,\mathrm{U}\, \phi) \mid \mathrm{EX}\phi \mid \mathrm{EG}\phi \mid \mathrm{E}(\phi \,\mathrm{U}\, \phi)$$

where a is an atomic proposition, describing properties of individual program states and $l : a$ is an atomic proposition that holds at program point l. The universal (A) and the existential (E) quantifiers allow expressing properties of all or some program executions starting in a state. The next temporal operator (X) allows expressing properties about the next state in a program execution. The globally operator (G) allows expressing properties that always (i.e., for all states) hold in a program execution. The until temporal operator (U) allows expressing a property that must hold eventually in a program execution, and another that must always hold until then. For instance, the robust reachability of the error in Figure 1 is expressed by the formula $\mathrm{A}(\mathrm{true}\, U\, \{l_5 : \mathrm{true}\})$, which means that for every execution (operator A) starting at the beginning of the program, the sequence of program states encountered during the execution must start with arbitrary states (formula true) followed ultimately (operator U) with a state at program location l_5 (formula $\{l_5 : \mathrm{true}\}$). In the following, we denote the robust reachability as $\mathrm{AF}\phi$, which is a shortcut for $\mathrm{A}(\mathrm{true}\,\mathrm{U}\,\phi)$, meaning that every program execution eventually reaches a state satisfying ϕ, i.e., $\mathrm{AF}\{l_5 : \mathrm{true}\}$ for Figure 1.

The semantics of CTL formulas is formally given by a satisfaction relation $\models$ between program states and CTL formulas. We refer to [1] for its formal definition. We write $s \models \phi$ if and only if the CTL formula ϕ holds for (the program executions starting in) the state $s \in \Sigma$.

3 Static Analysis of CTL Properties

FuncTion-V builds upon the static analysis framework proposed by Urban et al. [20]. Following the abstract interpretation theory [5], they derive a *sound and complete* semantics for proving a CTL property. Specifically, this semantics is a partial function $\Lambda_\phi : \Sigma \rightharpoonup \mathbb{O}$ (where $\mathbb{O}$ is the set of ordinals) defined only over the program states $s \in \Sigma$ that satisfy the CTL property ϕ, i.e., $s \models \phi$ if and only if $s \in \mathrm{dom}(\Lambda_\phi)$ [20, Theorem 1]. For instance, at the beginning of the program in Figure 1, $\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}$ is defined as follows:

$$\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}(s) = \begin{cases} v_1 & s(y) \leq s(z) \\ v_2 & s(x) > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

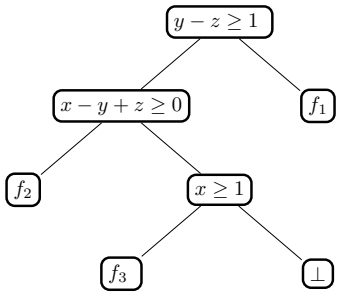


Figure 2: Decision tree inferred for $\mathrm{AF}\{l_5 : \mathrm{true}\}$ at the beginning of the program in Figure 1.

where $s(x)$ is value of the variable x in the state $s \in \Sigma$, and $v_1, v_2 \in \mathbb{O}$, meaning that $\mathrm{AF}\{l_5 : \mathrm{true}\}$ is satisfied in s if and only if $s(y) \leq s(z)$ or $s(x) > 0$. The concrete value of v_1 and v_2 is not relevant in the context of this paper so we do not discuss it (see [20] for further details).

A further *sound* abstract semantics suitable for static program analysis is derived leveraging the decision tree abstract domain $\mathcal{T}$, which is a numerical abstract domain based on piecewise-defined functions [17, 16]. Specifically, the abstraction $\Lambda^\natural_\phi \in \mathcal{T}$ of Λ_ϕ is a piecewise-defined partial function represented as a *decision tree*.

Figure 2 shows a decision tree abstracting the function $\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}$ shown above. The decision nodes are labeled by linear constraints (of the form $c_1 * X_1 + ... + c_k * X_k \geq c_{k+1}$) and recursively partition the space of possible values of the program variables: each constraint in a node is satisfied by the left subtree, while its negation is satisfied by the right subtree. The leaf nodes represent the value of the function corresponding to each partition. They are labeled by functions of the program variables (f_1, f_2, f_3 in Figure 2), or the special elements $\bot$ or $\top$, which explicitly represent undefined functions. The domain of the function represented by the decision tree in Figure 2 coincides with that of $\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}$ shown above. The partitioning is determined dynamically by the static analysis: partitions are modified by assignments and split (i.e., added) by boolean conditions and when merging control flows. A widening operator [4] extrapolates the function value in each partition and limits the size of the decision trees (and thus the number of partitions) to minimize the cost of the analysis and enforce its termination.

Let $\gamma \colon \mathcal{T} \to (\Sigma \rightharpoonup \mathbb{O})$ be the concretization function mapping decision trees to piecewise-defined partial functions. We refer to [17, 16] for its formal definition. The abstract semantics $\Lambda^\natural_\phi$ *under-approximates* the domain of Λ_ϕ (i.e., $\mathrm{dom}(\Lambda^\natural_\phi) \subseteq \mathrm{dom}(\gamma(\Lambda_\phi))$), and *over-approximates* its value (i.e., $\forall s \in \mathrm{dom}(\gamma(\Lambda^\natural_\phi)) \colon \Lambda_\phi(s) \leq \gamma(\Lambda^\natural_\phi)(s)$). In this way, $\Lambda^\natural_\phi$ yields sufficient preconditions for a state $s \in \Sigma$ to satisfy the CTL property ϕ: if $s \in \mathrm{dom}(\gamma(\Lambda^\natural_\phi))$ then $s \models \phi$ [20, Theorem 2]. In Figure 2, the value of $\Lambda_{\mathrm{AF}\{l_5:\mathrm{true}\}}$ is indeed over-approximated when $x \geq 1$, while $\mathrm{dom}(\gamma(\Lambda^\natural_\phi))$ coincides with $\mathrm{dom}(\Lambda_\phi)$ so the alarm when $y > z$ and $x \leq 0$ (where $\Lambda^\natural_\phi$ has value $\bot$ and is thus undefined) is a true positive (but that is not always the case in general).

4 Vulnerable Variable Identification

Let $\mathbb{X}$ denote the set of all program variables. We formally define when a set of variables $\mathcal{S} \subseteq \mathbb{X}$ is *safe* with respect to a CTL property ϕ. Given a variable $x \in \mathbb{X}$ and a value $v \in \mathbb{Z}$, we write $s[x \leftarrow v]$ for the program state obtained by writing the value v into the variable x in $s \in \Sigma$.

Definition 4.1 (**Safe Variables**). A set $\mathcal{S} = \{x_0 ... x_{n-1}\} \subseteq \mathbb{X}$ of program variables is *safe* with respect to a set of program states $D \subseteq \Sigma$, written $\textsc{safe}(\mathcal{S}, D)$, when there exists a program state $s \in D$ such that $\forall v_0, \ldots, v_{n-1} \in \mathbb{Z}^n$ we have $s[x_0 \leftarrow v_0, \ldots, x_{n-1} \leftarrow v_{n-1}] \in D$.

In other words, there is at least one program state that cannot escape D independently of the values of the safe variables. In the following, given a set of variables $\mathcal{S} \subseteq \mathbb{X}$ and a set of program states $D \subseteq \Sigma$, we write $D|_{\mathcal{S}}$ for the subset of all program states in D that remain in D independently of the values of the variables in $\mathcal{S}$: $D|_{\mathcal{S}} \stackrel{\text{def}}{=} \{s \in D \mid \forall v_0, \ldots, v_{n-1} \in \mathbb{Z}^n : s[x_0 \leftarrow v_0, \ldots, x_{n-1} \leftarrow v_{n-1}] \in D\}$. Given a CTL property ϕ, we abuse notation and write $\textsc{safe}(\mathcal{S}, \phi)$ to denote $\textsc{safe}(\mathcal{S}, \{s \in \Sigma \mid s \models \phi\})$. Note that safe variable sets are closed under inclusion, i.e., any subset of a safe variable set remains a safe variable set. We write $\overline{\textsc{safe}}(\mathcal{S}, D)$ when $\mathcal{S}$ is inclusion-wise *maximal*, i.e., $\textsc{safe}(\mathcal{S}, D) \wedge \forall \mathcal{S}' \supsetneq \mathcal{S} : \neg\textsc{safe}(\mathcal{S}', D)$. Inclusion-wise *maximal* sets are not unique. For example, for the program in Figure 1, we have $\overline{\textsc{safe}}(\{x\}, \text{AF}\{l_5 : \text{true}\})$ as well as $\overline{\textsc{safe}}(\{y, z\}, \text{AF}\{l_5 : \text{true}\})$.

Vulnerable variables are all variables that do not belong to a maximal set of safe variables.

Definition 4.2 (**Vulnerable Variables**). A set $\mathcal{V} \subseteq \mathbb{X}$ of program variables is *vulnerable* with respect to a set of program states $D \subseteq \Sigma$, written $\textsc{vulnerable}(\mathcal{V}, D)$, when $\overline{\textsc{safe}}(\mathbb{X} \setminus \mathcal{V}, D)$.

Namely, vulnerable variable sets are the *minimal* sets of variables the values of which must be controlled to remain in D, independently of the other (safe) variable values. Given a CTL property ϕ, we write $\textsc{vulnerable}(\mathcal{S}, \phi)$ to denote $\textsc{vulnerable}(\mathcal{S}, \{s \in \Sigma \mid s \models \phi\})$. Note that a set of variables $\mathcal{X} \subseteq \mathbb{X}$ could be *both* vulnerable and safe, if both $\overline{\textsc{safe}}(\mathcal{X}, D)$ and $\overline{\textsc{safe}}(\mathbb{X} \setminus \mathcal{X}, D)$ hold. In this case, it means that it is sufficient to control either $\mathcal{X}$ or $\mathbb{X} \setminus \mathcal{X}$ to remain in D despite the values of the other variables. This is the case for the program in Figure 1 as we have $\textsc{vulnerable}(\{y, z\}, \text{AF}\{l_5 : \text{true}\})$ (and $\overline{\textsc{safe}}(\{x\}, \text{AF}\{l_5 : \text{true}\})$) as well as $\textsc{vulnerable}(\{x\}, \text{AF}\{l_5 : \text{true}\})$ (and $\overline{\textsc{safe}}(\{y, z\}, \text{AF}\{l_5 : \text{true}\})$). Indeed, to ensure robust reachability of the error location, it is enough to control the values of y and z (such that $y \leq z$), or to control the value of x (such that $x > 0$). It is interesting to infer *all* vulnerable variables sets as this provides more information and control to the user to choose the more convenient way to achieve the same result.

Vulnerable Variable Concrete Semantics. We observe that, since the (domain of the) program semantics $\Lambda_\phi : \Sigma \rightharpoonup \mathbb{O}$ for a CTL property ϕ is a sufficient and necessary condition for satisfying ϕ [20, Theorem 1] (cf. Section 3), we can abstract Λ_ϕ to a *vulnerable variable semantics* $\mathcal{X}_\phi$ that identifies the vulnerable sets of program variables with respect to ϕ.

Definition 4.3 (**Vulnerable Variable Semantics**). Let $\Lambda_\phi : \Sigma \rightharpoonup \mathbb{O}$ be the program semantics for a CTL property ϕ. The *vulnerable variable semantics* $\mathcal{X}_\phi \in \mathcal{P}(\mathcal{P}(\mathbb{X}))$ for ϕ is defined as $\mathcal{X}_\phi \stackrel{\text{def}}{=} \alpha_\phi^{\mathcal{X}}(\Lambda_\phi)$, where $\alpha_\phi^{\mathcal{X}}(f) \stackrel{\text{def}}{=} \{\mathcal{V} \subseteq \mathbb{X} \mid \textsc{vulnerable}(\mathcal{V}, \text{dom}(f))\}$.

For the program in Figure 1, $\mathcal{X}_\phi = \{\{y, z\}, \{x\}\}$.

We have that just controlling the values of any vulnerable variable set in $\mathcal{X}_\phi$ yields a sufficient condition for satisfying the CTL property ϕ.

Theorem 4.1. *Let $\mathcal{V} \in \mathcal{X}_\phi$. The program executions starting in any program state $s \in dom(\Lambda_\phi)|_{\mathbb{X} \setminus \mathcal{V}}$ satisfy the CTL property ϕ.*

For the program in Figure 1, $\{y, z\} \in \mathcal{X}_\phi$ yields the sufficient condition $\{s \in \Sigma \mid s(y) \leq s(z)\}$ (i.e., $y \leq z$) and $\{x\} \in \mathcal{X}_\phi$ yields $\{s \in \Sigma \mid s(x) > 0\}$ (i.e., $x > 0$).

Algorithm 1 Safe Variables Identification

```
1:  function DEFINITELY-SAFE(𝕏, t, S)
2:      R ← ∅
3:      for x ∈ 𝕏 do
4:          C ← S ∪ {x}
5:          if CLOSED(C) then R ← R ∪ {C}
6:          else
7:              if ¬BLOCKED(C) then
8:                  t′ ← FORGET(t, {x})
9:                  if UNDEFINED(t′) then BLOCK(C)
10:                 else
11:                     E ← UNCONSTRAINED(t′)
12:                     R′ ← DEFINITELY-SAFE(𝕏 \ E, t′, C ∪ E)
13:                     R ← R ∪ R′
14:     if R = ∅ then
15:         CLOSE(S)
16:         R = {S}
17:     return R
```

Vulnerable Variable Abstract Semantics. The vulnerable variable semantics $\mathcal{X}_\phi$ is not computable, so we leverage $\Lambda^\natural_\phi$ (cf. Section 3) to derive a sound abstraction $\mathcal{X}^\natural_\phi$.

Let $t \in \mathcal{T}$ be a decision tree representing a piecewise-defined partial function $f : \Sigma \rightharpoonup \mathbb{O}$. We observe that, if we havoc the value of a variable $x \in \mathbb{X}$ in t (e.g., we set x to a non-deterministic random value), and the resulting decision tree t' still represents a partially defined function (i.e., t' still contains leaves not labeled with $\bot$ or $\top$), then the variable x is *safe* with respect to $\mathrm{dom}(f)$ (cf. Definition 4.1). This process can be continued iteratively for each program variable, until obtaining (in general, an *under-approximation* of) a maximal safe variable set.

The function DEFINITELY-SAFE in Algorithm 1 considers all subsets of program variables to determine which ones are safe, and returns the set of all maximal safe variables sets it can find. Havoc is performed by the function FORGET invoked at Line 8 [16, Section 5.2.3]. If the resulting tree t' represents a totally undefined function (cf. Line 9), it means that the current candidate set C of variables (cf. Line 4) is not safe and so the recursive iteration should be stopped. The set C is *blocked* (cf. Line 9) to shortcut future iterations with candidate sets that are supersets of C, since they cannot be safe either (cf. Line 7). Let us consider the decision tree in Figure 2. The tree resulting after the call to FORGET$(\cdot, \{x\})$ is depicted in Figure 3. Further attempts to havoc any of the other variables result in the blocked sets $\{x, y\}$ and $\{x, z\}$. The definitely safe variable set $\{x\}$ is thus *closed* (cf. Line 15) and added to the returned result set (cf. Line 16). The decision tree resulting from either FORGET$(\cdot, \{y\})$ or FORGET$(\cdot, \{z\})$ is shown in Figure 4. Either call results in the simultaneous havoc of the other variable. Line 11 thus collects all variables that are unconstrained in t' (i.e., y and z) and adds them to the candidate definitely safe variable set in the recursive call (cf. Line 12). The call results in closing the set $\{y, z\}$ because a further havoc of x is shortcut since $\{x, y\} \subseteq \{y, z\} \cup \{x\}$ is blocked. Further iterations encountering already closed sets are also shortcut (cf. Line 5) and so the result returned by DEFINITELY-SAFE is $R = \{\{x\}, \{y, z\}\}$.

We can now define the potentially vulnerable variable semantics $\mathcal{X}^\natural_\phi$.

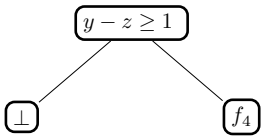


Figure 3: Figure 2 after FORGET$(\cdot, \{x\})$.

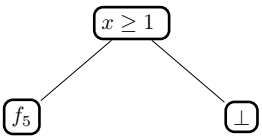


Figure 4: Figure 2 after FORGET$(\cdot, \{w\})$, $w \in \{y, z\}$.

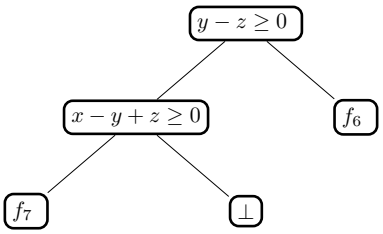


Figure 5: Decision tree inferred for the program in Figure 1 without CDA.

Definition 4.4 (**Potentially Vulnerable Variable Semantics**)**.** Let $\Lambda^{\natural}_{\phi} \in \mathcal{T}$ be the abstract program semantics for a CTL property ϕ. The *potentially vulnerable variable semantics* $\mathcal{X}^{\natural}_{\phi} \in \mathcal{P}(\mathcal{P}(\mathbb{X}))$ for ϕ is defined as $\mathcal{X}^{\natural}_{\phi} \stackrel{\text{def}}{=} \{\mathbb{X} \setminus \mathcal{S} \mid \mathcal{S} \in \text{DEFINITELY-SAFE}(\Lambda^{\natural}_{\phi}, \mathbb{X})\}$.

For any $\mathcal{W} \in \mathcal{X}^{\natural}_{\phi}$ there exists $\mathcal{V} \in \mathcal{X}_{\phi}$ such that $\mathcal{V} \subseteq \mathcal{W}$. Thus, we lose the minimality of vulnerable variable sets. Nonetheless, controlling the values of the potentially vulnerable variables still yields a sufficient condition for satisfying the CTL property ϕ.

Theorem 4.2. *Let $\mathcal{V} \in \mathcal{X}_{\phi}$. The program executions starting in any program state $s \in dom(\gamma(\text{FORGET}(\Lambda^{\natural}_{\phi}, \mathbb{X} \setminus \mathcal{V}))$ satisfy the CTL property ϕ.*

5 Conflict-Driven CTL Analysis

In order to improve the precision of the analysis, we have equipped FUNCTION-V with a conflict-driven analysis (CDA) [7], generalized to handle all program properties expressed in CTL (and not just program termination as in [7]). Following an imprecise CTL analysis, the inferred decision tree defines a partition of the program states where the property is verified (tree leaves labeled with defined functions) and where the analysis is inconclusive (tree leaves labeled with $\bot$ or $\top$). The partitions corresponding to the undefined leaves of the tree form the *conflicting domain*. For instance, Figure 5 shows another possible decision tree abstraction of $\Lambda_{\text{AF}\{l_5:\text{true}\}}$ for the program in Figure 1 (less precise than that in Figure 2). The conflicting domain in this case is $y - z \geq 0 \wedge x < y - z$. Drawing inspiration from constraint programming [21], we split the conflicting domain using various cutting heuristic (e.g., splitting on the variable with the largest range of values in the conflicting domain), and repeat the CTL analysis on each resulting partition. For the tree in Figure 5, we split on x and repeat the analysis with $x \leq 0$ and $x > 0$. The process proceeds recursively until the CTL property is verified, or a threshold is exceeded. This yields the tree in Figure 2, for which FUNCTION-V precisely finds the vulnerable variable sets $\{y, z\}$ and $\{x\}$. Instead, with the decision tree in Figure 5, FUNCTION-V only finds $\{y, z\}$.

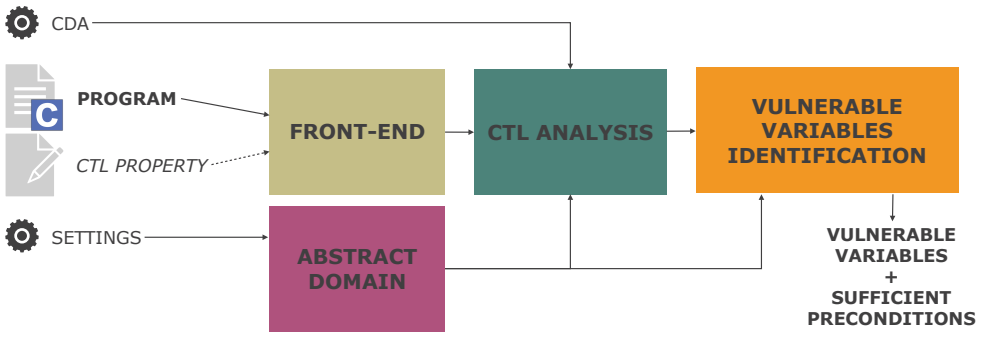


Figure 6: Overview of FuncTion-V's architecture.

6 Tool Architechture

FuncTion-V is implemented in OCaml. Figure 6 shows an overview of its architecture.

The tool takes as input a C program and a CTL property of interest (cf. Section 2). The front-end takes care of parsing the program (using an ad-hoc LR parser generated using Menhir[1]), building its internal representation, and passing it to the CTL analysis engine (cf. Section 3). The decision tree abstract domain (cf. Section 3) used by the CTL analysis can be configured (SETTINGS in Figure 6) to use different (more or less precise) domain and value representations [17] provided by the APRON numerical abstract domain library [10], as well as different (more or less sophisticated) widening heuristics [4]. The CTL analysis can optionally be refined using CDA (cf. Section 5, CDA in Figure 6).

The resulting decision tree is finally passed to the vulnerable variables identification analysis (cf. Section 4), which outputs the potentially vulnerable variables sets found and their corresponding sufficient preconditions for satisfying the given CTL property of interest.

7 Experimental Evaluation

We evaluated our tool on a number of test cases obtained from the literature [17, 18, 4, 20, 3, 6] and from the termination and reachSafety categories of the 11th International Competition on Software verification (SV-COMP 2023). The experiments were conducted on a 64-bit 8-Core CPU (AMD® Ryzen 7 pro 5850u) with 16 GB of RAM on Ubuntu 20.04 with a timeout of 120s. We configured the analysis to use decision tree with polyhedral constraints for all tests, and we evaluate the impact of the CDA extension on its precision.

For each test case, FuncTion-V found between zero and three potentially vulnerable variables sets. We present in Figure 7 the number of programs for which FuncTion-V found no potentially vulnerable set (i.e., the property of interest always holds), and for which there are n potentially vulnerable sets (i.e., one of these set must be controlled to ensure the property). We additionally detail the total (for all programs) and average (for each program) analysis time, and the average number of lines of code. When the CTL analysis returns a totally undefined function (i.e., a decision tree with only $\bot$ or $\top$ leaves), FuncTion-V returns a single potentially vulnerable variable set containing all program variables, but no sufficient condition can be found to ensure the property of interest. These cases are indicated in parentheses in Figure 7.

[1] https://gallium.inria.fr/~fpottier/menhir/

	Vulnerable Sets	Number of Programs	Total Time	Average Time	Average LOC
no-CDA	0	146	15 s	0.1 s	79
	1	165 (+15)	21 s	0.1 s	22
	2	30	10 s	0.33 s	24
	3	6	5 s	0.7 s	25
CDA	0	154	23 s	0.15 s	77
	1	160 (+14)	934 s	5 s	22
	2	79	432 s	15 s	25
	3	5	124 s	25 s	26

Figure 7: Number of potentially vulnerable variable sets.

	Termination	Robust Reachability	CTL
no-CDA	17%	26%	32%
CDA	16%	22%	28%

Figure 8: Average minimum percentage of potentially vulnerable variables.

In Figure 8, tests are categorized based on the property of interest: *termination* (242 tests), *robust reachability* (95 tests), and *CTL* (130 tests) for other CTL properties. For each category, we report the average minimum percentage of (potentially vulnerable) variables that FuncTion-V identified (excluding the cases in which no sufficient condition can be found).

Figure 7 and Figure 8 show that CDA improves the precision of FuncTion-V, allowing it to identify *fewer* and *smaller* potentially vulnerable variables sets. This precision improvement comes with a non-negligible (but still small on average) performance cost (cf. Figure 7).

The full experimental results for each test case can be found as part of our artifact[2].

8 Related Work

The program property of *robust reachability* [8, 9] refines the standard notion of reachability of a bug given a partition of the program variables in a *controlled* and an *uncontrolled* set: a bug is robustly reachable if it is reachable whatever the values of the uncontrolled variables. FuncTion-V can analyze robust reachability properties expressible as a CTL formula of the form AFϕ. The vulnerable variables sets identified by FuncTion-V identify the controlled program variables, and the corresponding sufficient preconditions inferred by FuncTion-V yield the space of values for the *controlled* program variables that ensure the robust reachability property, i.e., instead of returning a single bug witness like the symbolic execution framework proposed by [8, 9], FuncTion-V returns a set of bug witnesses. Instead, the approach recently proposed by Sellami et al. [15] infers constraints on the *uncontrolled* program variables.

The non-exploitability analysis proposed by Parolini and Miné [14] uses a combination of forward taint and value analyses by abstract interpretation to infer the absence of (potentially) exploitable runtime errors. A runtime error is (potentially) exploitable if it is triggered by a subset of the variables initialized by an external *input* (that models a potential attacker). Our tool is able to prove non-exploitability by analyzing the absence of (runtime) errors (as a CTL property). If the *input* variables are safe then the runtime errors are non-exploitable (i.e., there is no choice of values for the input variables that ensures the presence of runtime errors).

[2] https://zenodo.org/records/10964500

The under-approximating analysis proposed by Miné [13] (recently extended in [12]) aims at finding sufficient preconditions for (un)desired program properties. It is used by the CTL analysis in FuncTion-V to handle existential CTL properties. We could also use it by itself, as an alternative to the CTL analysis, before the vulnerable variable identification. In this case, however, we would lose the sensitivity to termination, i.e., the (un)desired program property is satisfied provided that the relevant program point(s) are reached by the program execution.

9 Conclusion and Future Work

In this work, we have presented a new static analyzer called FuncTion-V for the automatic identification of vulnerable program variables that could be controlled to ensure an (un)desirable CTL property. We have derived our static analysis within the framework of abstract interpretation, building upon a static analysis for CTL program properties that we extended with a conflict-driven abstraction refinement-style analysis. Using experimental evidence on a wide variety of benchmarks, we showed that FuncTion-V is able to discover vulnerable variables and the corresponding sufficient conditions to ensure the CTL property of interest. It remains for future work to extend the analysis to support a broader range of programs (e.g., array- and pointer-manipulating programs). We also plan to investigate a combination of the non-exploitability analysis of Parolini and Miné [14] with our CTL and vulnerable variable analysis. Finally, we would like to study the applicability of our approach for identifying relevant input features in the context of machine learning explainability [11].

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2008.

[3] Byron Cook, Eric Koskinen, and Moshe Y. Vardi. Temporal Property Verification as a Program Analysis Task. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2011.

[4] Nathanaël Courant and Caterina Urban. Precise Widening Operators for Proving Termination by Abstract Interpretation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 136–152, 2017.

[5] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[6] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness Modulo Theory: A New Approach to LTL Software Model Checking. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 49–66. Springer, 2015.

[7] Vijay D'Silva and Caterina Urban. Conflict-Driven Conditional Termination. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2015.

[8] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 669–693. Springer, 2021.

[9] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Introducing Robust Reachability. *Formal Methods in System Design*, November 2022.

[10] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[11] João Marques-Silva, Thomas Gerspacher, Martin C. Cooper, Alexey Ignatiev, and Nina Narodytska. Explanations for monotonic classifiers. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 7469–7479. PMLR, 2021.

[12] Marco Milanese and Antoine Miné. Generation of Violation Witnesses by Under-Approximating Abstract Interpretation. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part I*, volume 14499 of *Lecture Notes in Computer Science*, pages 50–73. Springer, 2024.

[13] Antoine Miné. Backward Under-Approximations in Numeric Abstract Domains to Automatically Infer Sufficient Program Conditions. *Sci. Comput. Program.*, 93:154–182, 2014.

[14] Francesco Parolini and Antoine Miné. Sound Abstract Nonexploitability Analysis. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*, volume 14500 of *Lecture Notes in Computer Science*, pages 314–337. Springer, 2024.

[15] Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé, and Sébastien Bardin. Inference of robust reachability constraints. *Proc. ACM Program. Lang.*, 8(POPL):2731–2760, 2024.

[16] Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs*. PhD thesis, 2015.

[17] Caterina Urban and Antoine Miné. A Decision Tree Abstract Domain for Proving Conditional Termination. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014.

[18] Caterina Urban and Antoine Miné. Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, volume 8931 of *Lecture Notes in Computer Science*, pages 190–208. Springer, 2015.

[19] Caterina Urban and Antoine Miné. Inference of ranking functions for proving temporal properties by abstract interpretation. *Comput. Lang. Syst. Struct.*, 47:77–103, 2017.

[20] Caterina Urban, Samuel Ueltschi, and Peter Müller. Abstract Interpretation of CTL Properties. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*,

pages 402–422. Springer, 2018.

[21] Ghiles Ziat. *A Combination of Abstract Interpretation and Constraint Programming*. PhD thesis, 2019.

[u4] Naïm Moussaoui Remil, **CU**

Termination Resilience Static Analysis

Under Submission

Termination Resilience Static Analysis

Naïm Moussaoui Remil and Caterina Urban

Inria & ENS | PSL, Paris, France
{naim.moussaoui-remil,caterina.urban}@inria.fr

Abstract. We present a novel abstract interpretation-based static analysis framework for proving Termination Resilience, the absence of Robust Non-Termination vulnerabilities in software systems. Robust Non-Termination characterizes programs where an externally-controlled input can force infinite execution, independently of other uncontrolled variables. Our framework is a semantic generalization of Cousot and Cousot's abstract interpretation-based ranking function derivation, and our sound static analysis extends Urban and Miné's decision tree abstract domain in a non-trivial way to manage the distinction between controlled and uncontrolled program variables. Our approach is implemented in an open-source tool and evaluated on benchmarks sourced from SV-COMP and modeled after real-world software, demonstrating practical effectiveness in verifying Termination Resilience and detecting potential Robust Non-Termination vulnerabilities.

1 Introduction

Software bugs vary significantly in their severity and impact. While some cause minor glitches or inefficiencies, others pose serious risks — particularly those that can be exploited by an external adversary [15,27] (e.g., attacker, scheduler, etc.). Recognizing and prioritizing such vulnerabilities is essential for ensuring software safety and security. Bugs that cause *non-termination* can be weaponized by a malicious actor to lock software into an endless cycle of execution leading to system slowdowns, resource exhaustion, and denial-of-service conditions. Identifying and fixing these bugs is critical for the reliability of software systems. Several approaches have been proposed in the literature to prove termination or to identify diverging executions. However, these typically assume a worst-case setting, treating all sources of non-determinism as controlled by an adversary. This often results in overly conservative analyses yielding an excessive number of alarms. In contrast, in this work, we propose a more nuanced approach for reasoning about termination; one that explicitly distinguishes between *externally-controlled (potentially malicious)* and *uncontrolled (benign)* sources of non-determinism.

We introduce and formalize the *Robust Non-Termination* program vulnerability, to characterize programs for which there exists an *externally-controlled input* value that forces their execution into an infinite loop, independently of the values of other *uncontrolled variables* (e.g., random seeds, etc.). For instance, let us consider the Decode function in Figure 1, a decoder for JPEG images

```
1  function decode(jpegData){
2   frame = parse(jpegData)
3   var maxH = 0, maxV = 0;
4   for (component in frame.components) {
5      if( maxH < component.h) maxH = component.h;
6      if( maxH < component.v) maxH = component.v;
7   }
8   var mcusRow = Math.ceil(8 / maxH);
9   var mcusCol = Math.ceil(8 / maxV);
10  mcuExcepted = mcusRow * mcusCol;
11  mcusolved = 0;
12  while( mcusolved < mcuExcepted){
13     //...
14     mcusolved++;
15  }
16 }
```

Fig. 1: Javascript program from the `jpeg-js` library vulnerable to Robust Non-Termination. `Math.ceil(x)` returns the smallest integer greater or equal to x.

from the widely used `jpeg-js` Javascript library. It takes an input a buffer of bytes encoding a JPEG image, and parses it into a frame (an internal image representation). It then updates the values of the variables `maxH` and `maxV` by iterating on the field `components` of the frame (cf. Lines 3-7). However, if the input buffer leads to a frame with an empty list of components, the values of `maxH` and `maxV` will never be updated and the division at Line 8 and 9 will return Infinity. Subsequently, `mcuExcepted` at Line 10 will also have value Infinity and the loop condition at Line 12 will then always be true. Thus, Decode is vulnerable to Robust Non-Termination. This is a high severity vulnerability[1] as a denial of service condition can be triggered by a carefully chosen buffer value.

We design an abstract interpretation-based static analysis framework to prove *Termination Resilience* — the absence of Robust Non-Termination vulnerabilities. Termination Resilience ensures that, for all *externally-controlled input*, there exists a terminating program execution – meaning that the program can potentially *resist* non-termination exploits. Our framework is a more expressive semantic generalization of Cousot and Cousot's idea of computing a ranking function by abstract interpretation [12]. Specifically, we extend their semantic definitions to handle both demonic and angelic non-determinism, distinguishing cases where the program execution is influenced by externally-controlled inputs from those where it is not. We derive a sound static analysis for Termination Resilience by building upon Urban and Miné's decision tree abstract domain [33], enhancing it with novel and non-trivial transformer operators to effectively manage the distinction between controlled and uncontrolled program variables. Our static analysis automatically infers sufficient preconditions that

[1] https://www.cve.org/CVERecord?id=CVE-2022-25851

```
1 x := input;
2 z := 10;
if 3(...) {
   while 4(z >= 0) {
     5z := z - x;
   }
} else {
  6c := [-2, 1];
   while 7(z >= x) {
     8z := z + c;
   }
}9
```

Fig. 2: Program P.

ensure Termination Resilience for a program. Consider program P in Figure 2 – inspired from our benchmarks and written in the small programming language that we use for illustration in the paper: the program variable `x` is an externally-controlled input (cf. program label 1), while the value of `c` is non-deterministically chosen between -2 and 1 (cf. program label 7). Both `while` loops in the program are non-terminating, the first when $x \leq 0$, and the second when ($z \geq x$ and) $c \geq 0$. Our static analysis raises an alarm only for the first loop, since its (non-)termination depends on externally-controlled variables: the analysis automatically infers the precondition $x > 0$ for termination.

Our analysis is implemented in an open-source tool and evaluated on benchmarks sourced from the International Competition on Software Verification (SV-COMP) and modeled after real-world software. The experimental results demonstrate the practical effectiveness of the analysis for verifying Termination Resilience, and detecting (potential) Robust Non-Termination vulnerabilities. Notably, our approach substantially reduces the number of alarms compared to traditional termination analysis, helping prioritize warnings based on exploitability.

2 Termination Resilience

In this section, we give a mathematical characterization of the behavior of a program, and we formally define our property of interest, *Termination Resilience*.

Program Semantics. We model the operational behavior of a program as a *transition system* $\langle \Sigma, \tau \rangle$ where Σ is a (potentially infinite) set of program states, and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes possible transition between states [10,9]. Specifically, for our purposes, $\tau \triangleq \tau^i \uplus \tau^r$ is the disjoint union of *input transitions* in τ^i, which are transitions that consume an external input, and *regular transitions* in τ^r. For simplicity, without loss of generalization, we assume that all transitions $\langle s, s' \rangle \in \tau$ from a state $s \in \Sigma$ are of the same kind,

(a) The transition system $\langle \Sigma_1, \tau_1 \rangle$. (b) The transition system $\langle \Sigma_2, \tau_2 \rangle$.

Fig. 3: The transition systems from Example 1 and 2. Input transitions are represented with dashed red lines, while solid black lines represent regular transitions.

i.e., either all input transition or all regular transitions. The set of final program states is $\Omega_\tau \triangleq \{s \in \Sigma \mid \forall s' \in \Sigma \colon \langle s, s' \rangle \notin \tau\}$.

Given a transition system $\langle \Sigma, \tau \rangle$, the function $\mathrm{pre}_{\tau^r} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a given set of states $X \subseteq \Sigma$ to the set of their predecessors with respect to τ^r: $\mathrm{pre}_{\tau^r}(X) \triangleq \{s \in \Sigma \mid \exists s' \in X : \langle s, s' \rangle \in \tau^r\}$, and the function $\widetilde{\mathrm{pre}}_{\tau^i} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a set of states $X \subseteq \Sigma$ to the set of states whose successors with respect to τ^i are all in X: $\widetilde{\mathrm{pre}}_{\tau^i}(X) \triangleq \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \in \tau^i \Rightarrow s' \in X\}$.

In the following, given a set S, let ε be an empty sequence of elements in S, S^+ be the set of all non-empty finite sequences of elements in S, S^ω be the set of all infinite sequences, $S^{+\infty} \triangleq S^+ \cup S^\omega$ be the set of all non-empty finite or infinite sequences of elements in S, and $S^{*\infty} \triangleq S^{+\infty} \cup \{\varepsilon\}$. We write $T^+ \triangleq T \cap S^+$, $T^\omega \triangleq T \cap S^\omega$ for the selection of the non-empty finite sequences and the infinite sequences of $T \in \mathcal{P}(S^{+\infty})$, and $T; T' = \{tst' \in S^{+\infty} \mid s \in S \wedge ts \in T \wedge st' \in T'\}$ for the merging of two sets of sequences $T \in \mathcal{P}(S^+)$ and $T' \in \mathcal{P}(S^{+\infty})$, when a finite sequence in T terminates with the initial element of a sequence in T'.

A *trace* $\sigma \in \Sigma^{+\infty}$ is a non-empty sequence of program states where each pair of consecutive states $s, s' \in \Sigma$ is described by the transition relation, i.e., $\langle s, s' \rangle \in \tau$. We write σ_n to denote its n-th state of a trace $\sigma \in \Sigma^{+\infty}$ (with $n < |\sigma|$), σ^i (resp. σ^r) for the (possibly empty) sequence of input (resp. regular) transitions in σ, and T^i for the set $\{\sigma^i \mid \sigma \in T\}$ of sequences of input transitions in traces in a set $T \in \mathcal{P}(\Sigma^{+\infty})$ (resp. T^r for $\{\sigma^r \mid \sigma \in T\}$). The *trace semantics* $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition system $\langle \Sigma, \tau \rangle$ is the union of all non-empty finite traces terminating in Ω_τ, and all infinite traces. It can be expressed as a least fixpoint in the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$[9]:

$$\begin{aligned} \Lambda &= \mathsf{lfp}^{\sqsubseteq} \Theta \\ \Theta(T) &\triangleq \Omega_\tau \cup (\tau; T) \end{aligned} \tag{1}$$

where the computational order is $T_1 \sqsubseteq T_2 = T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$. In the following, we write $\Lambda[\![P]\!]$ to denote the trace semantics of a given program P. Given an initial set of states $I \subseteq \Sigma$, we define $\Lambda[\![P]\!]I \triangleq \{\sigma \in \Lambda[\![P]\!] \mid \sigma_0 \in I\}$ as the restriction of the trace semantics of P to traces starting in I.

Example 1. Let $\langle \Sigma_1, \tau_1 \rangle$ be a transition system with $\Sigma_1 = \{a, b, c\}$ and $\tau_1 = \tau_1^i \cup \tau_1^r$ where $\tau_1^i = \{\langle a, b \rangle, \langle a, c \rangle\}$ and $\tau_1^r = \{\langle c, c \rangle\}$ (cf. Figure 3a). The trace semantics generated by $\langle \Sigma_1, \tau_1 \rangle$ and starting in $I = \{a\}$ is $\Lambda_1 = \{ab, ac^\omega\}$. ◁

Example 2. Let $\langle \Sigma_1, \tau_1 \rangle$ be the transition system from Example 1 and let $\langle \Sigma_2, \tau_2 \rangle$ be another transition system with $\Sigma_2 = \Sigma_1 \cup \{d, e\}$ and $\tau_2 = \tau_1^i \cup \tau_2^r$ where $\tau_2^r = \tau_1^r \setminus \{\langle c, c\rangle\} \cup \{\langle b, d\rangle, \langle b, e\rangle, \langle e, e\rangle\}$ (cf. Figure 3b). The trace semantics generated by $\langle \Sigma_2, \tau_2 \rangle$ and starting in $I = \{a\}$ is $\Lambda_2 = \{abd, abe^\omega, ac\}$. ◁

Program Property. As customary in abstract interpretation [10], we represent properties of entities in a universe $\mathbb{U}$ by a subset of this universe. We formally define the Robust Non-Termination program property.

Definition 1 (Robust Non-Termination). *The* Robust Non-Termination *program property is the set of sets of traces for which a (possibly empty) sequence of input transitions only yields diverging traces:*

$$\mathcal{RNT} \triangleq \left\{ T \in \mathcal{P}(\Sigma^{+\infty}) \mid \exists i \in T^i \, \forall \sigma \in T \colon \sigma^i = i \Rightarrow \sigma \in T^\omega \right\} \tag{2}$$

Note that any set $T \in \mathcal{P}(\Sigma^\omega)$ only containing infinite traces always belongs to $\mathcal{RNT}$ (in such case $\sigma^i = \varepsilon$ for all σ in T), i.e., an always non-terminating program is always vulnerable to non-termination exploits.

Example 3. Let $\langle \Sigma_1, \tau_1 \rangle$ be the transition system from Example 1. The input transition $\langle a, c\rangle$ only yields the diverging trace $ac^\omega \in \Lambda_1$. Thus $\Lambda_1 \in \mathcal{RNT}$. ◁

The negation of Robust Non-Termination is our property of interest. It expresses the ability of the program to (possibly) defend itself from non-termination exploits. Formally, we call this property *Termination Resilience*.

Definition 2 (Termination Resilience). *The* Termination Resilience *program property is the set of sets of program traces for which there exists a terminating trace for all sequence of input transitions:*

$$\mathcal{TR} \triangleq \neg\mathcal{RNT} = \{ T \in \mathcal{P}(\Sigma^{+\infty}) \mid \forall i \in T^i \, \exists \sigma \in T^+ \colon \sigma^i = i \} \tag{3}$$

Example 4. Let $\langle \Sigma_2, \tau_2 \rangle$ be the transition system from Example 2. For any sequence of input transitions in $(\Lambda_2)^i = \{\langle a, b\rangle, \langle a, c\rangle\}$, there is at least one terminating trace in Λ_2, i.e., abd for $\langle a, b\rangle$, and ac for $\langle a, c\rangle$. Thus, $\Lambda_2 \in \mathcal{TR}$. ◁

The trace semantics $\Lambda[\![P]\!]$ of a program P, fully characterizing the behavior of P, is sound and complete for verifying Termination Resilience (and, equivalently, disproving Robust Non-Termination), for an initial set of states $I \subseteq \Sigma$:

$$P \models_I \mathcal{TR} \Leftrightarrow \Lambda[\![P]\!]I \in \mathcal{TR} \Leftrightarrow \Lambda[\![P]\!]I \notin \mathcal{RNT} \Leftrightarrow P \not\models_I \mathcal{RNT}. \tag{4}$$

In the next two sections, we abstract the trace semantics Λ to a special-purpose sound and complete (but not computable) program concrete semantics Λ_{tr} that forgets details of the program behavior that are irrelevant for reasoning about Termination Resilience. Next, in Section 5, we further abstract this semantics into a sound but computable abstract semantics $\Lambda_{\mathsf{tr}}^\natural$ that yields a static analysis for Termination Resilience.

3 Termination Resilience Semantics

We derive our concrete semantics Λ_{tr} by abstract interpretation of the trace semantics Λ. The abstraction eliminates program traces that are potentially branching to non-termination through different externally-controlled inputs, and proves Termination Resilience for the remaining traces by associating a well-founded quantity [30,13] to program states belonging to terminating traces. Specifically, we abstract the trace semantics $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ into the *termination resilience semantics* $\Lambda_{\mathsf{tr}} \colon \Sigma \rightharpoonup \mathbb{O}$ (where $\mathbb{O}$ is the of ordinals), which is the best (potential) ranking function [12,31] for the program states in the traces in Λ.

We define the termination resilience abstraction $\alpha_{\mathsf{tr}} \colon \mathcal{P}(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ to map sets of program traces to a (potential) ranking function as:

$$\alpha_{\mathsf{tr}}(T) \triangleq \alpha_{\mathsf{v}} \circ \overrightarrow{\alpha}(T) \tag{5}$$

where $\overrightarrow{\alpha} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma \times \Sigma)$ extracts the smallest transition relation $r \subseteq \Sigma \times \Sigma$ that generates a given set of traces T: $\overrightarrow{\alpha}(T) \triangleq \{\langle s, s' \rangle \mid \exists \sigma, \sigma' \colon \sigma s s' \sigma' \in T\}$, and $\alpha_{\mathsf{v}} \colon \mathcal{P}(\Sigma \times \Sigma) \to (\Sigma \rightharpoonup \mathbb{O})$ ranks the elements in the domain of r:

$$\alpha_{\mathsf{v}}(\emptyset) \triangleq \dot{\emptyset}$$

$$\alpha_{\mathsf{v}}(r)s \triangleq \begin{cases} 0 & s \in \Omega_r \\ \mathsf{sup} \left\{ \alpha_{\mathsf{v}}(r)s' + 1 \,\middle|\, \langle s, s' \rangle \in r^i \right\} & s \in \widetilde{\mathrm{pre}}_{r^i}(\mathsf{dom}(\alpha_{\mathsf{v}}(r))) \\ \mathsf{inf} \left\{ \alpha_{\mathsf{v}}(r)s' + 1 \,\middle|\, \langle s, s' \rangle \in r^r \right\} & s \in \mathrm{pre}_{r^r}(\mathsf{dom}(\alpha_{\mathsf{v}}(r))) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{6}$$

with $\dot{\emptyset}$ representing the totally undefined function. The (potential) ranking function is defined incrementally, starting from the final states in Ω_r, where the function has value 0 (and is undefined elsewhere). Then, the domain of the function grows by retracing the program (transitions) backwards and counting the number of performed program steps as value of the function. We model the nondeterminism arising from input transitions *demonically*, requiring all successor states to already be in the domain of the function ($s \in \widetilde{\mathrm{pre}}_{r^i}(\mathsf{dom}(\alpha_{\mathsf{v}}(r)))$), and count the *maximum* number of performed program steps. Instead, nondeterminism arising from regular transitions is treated *angelically*, and the potential ranking functions counts the minimum number of steps to termination.

We can now formally define the termination resilience semantics Λ_{tr}.

Definition 3 (Termination Resilience Semantics). *Let $\langle \Sigma, \tau \rangle$ be a transition system. Its termination resilience semantics $\Lambda_{\mathsf{tr}} \in (\Sigma \rightharpoonup \mathbb{O})$ is:*

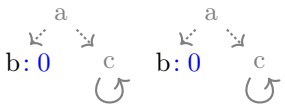
$$\Lambda_{\mathsf{tr}} \triangleq \alpha_{\mathsf{tr}}(\Lambda) = \mathsf{lfp}^{\sqsubseteq}_{\dot{\emptyset}} \Theta_{\mathsf{tr}}$$

$$\Theta_{\mathsf{tr}}(f) \triangleq \lambda s. \begin{cases} 0 & s \in \Omega_\tau \\ \mathsf{sup} \left\{ f(s') + 1 \,\middle|\, \langle s, s' \rangle \in \tau^i \right\} & s \in \widetilde{\mathrm{pre}}_{\tau^i}(\mathsf{dom}(f)) \\ \mathsf{inf} \left\{ f(s') + 1 \,\middle|\, \langle s, s' \rangle \in \tau^r \right\} & s \in \mathrm{pre}_{\tau^r}(\mathsf{dom}(f)) \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{7}$$

where $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ *is the trace semantics (cf. Equation 2), and the computational order* $f_1 \sqsubseteq f_2$ *is* $\mathsf{dom}(f_1) \subseteq \mathsf{dom}(f_2) \wedge \forall x \in \mathsf{dom}(f_1) : f_1(x) < f_2(x)$.

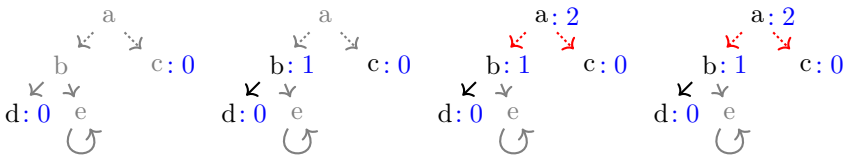
We write $\Lambda_{\mathsf{tr}}[\![P]\!]$ for the termination resilience semantics of a program P.

Example 5. Let $\langle \Sigma_1, \tau_1 \rangle$ and $\Lambda_{\mathsf{t}1}$ be the transition system from Example 1 and its termination resilience trace semantics, respectively. The fixpoint iterates of its termination resilience semantics $\Lambda_{\mathsf{tr}1}$ are the following:



where states in the domain of the function are mapped to their ranking function value, and states outside the domain are colored in gray. The fixpoint $\Lambda_{\mathsf{tr}1}$ is immediately reached and is only defined over the state b. ◁

Example 6. Let $\langle \Sigma_2, \tau_2 \rangle$ and $\Lambda_{\mathsf{t}2}$ be the transition system from Example 2 and its termination resilience trace semantics, respectively. The fixpoint iterates of its termination resilience semantics $\Lambda_{\mathsf{tr}2}$ are the following:



Note that $\Lambda_{\mathsf{tr}2}$ is defined over the state b even if a transition can lead to the diverging trace be^{ω} because the non-deterministic choice is treated angelically: there is at least one choice that ensures termination (via the trace bd). ◁

The termination resilience semantics $\Lambda_{\mathsf{tr}}[\![P]\!]$ of a program P is sound and complete for verifying Termination Resilience, for an initial set of states $I \subseteq \Sigma$:

Theorem 1 (Soundness&Completeness). $\Lambda[\![P]\!]I \in \mathcal{TR} \Leftrightarrow I \subseteq \mathsf{dom}(\Lambda_{\mathsf{tr}}[\![P]\!])$

Proof (Sketch). The proof uses Park's fixpoint induction principle, along the lines of the soundness and completeness proofs for the potential and definite termination semantics of Cousot and Cousot [12,31]. □

4 Denotational Termination Resilience Semantics

The formal treatment given in the previous section is language independent. In the following, for simplicity, we introduce a small sequential programming language that we use for illustration throughout the rest of the paper:

$$
\begin{aligned}
\text{AExp} \ni \mathsf{a} &::= x \mid c \mid -\mathsf{a} \mid \mathsf{a} \diamond \mathsf{a} \\
\text{Stmt} \ni \mathsf{s} &::= {}^{l}\,\mathsf{skip} \mid {}^{l}\,x := \mathsf{input} \mid {}^{l}\,x := [c_1, c_2] \mid {}^{l}\,x := \mathsf{a} \\
&\quad \mid \mathsf{if}\,{}^{l}\,\mathsf{a} \bowtie 0\,\{\mathsf{s}\} \mid \mathsf{while}\,{}^{l}\,\mathsf{a} \bowtie 0\,\{\mathsf{s}\} \mid \mathsf{s}; \mathsf{s} \\
\text{Prog} \ni \mathsf{p} &::= \mathsf{s}^{l}
\end{aligned}
$$

```
1 x := input;
2 z := 10;

while 3 (z >= 0) {
  4 z := z - x;
}5
```

(a) Program P_1

```
1 x = input;
2 z := 10;
3 c := [-2, 1];
while 4 (z >= x) {
  5 z := z + c;
}6
```

(b) Program P_2

Fig. 4: Programs akin to the transitions systems in Example 1 (a) and 2 (b).

where $x \in \mathbb{X}$ ranges over program variables, $c \in \mathbb{Z}$, $c_1 \in \mathbb{Z} \cup \{-\infty\}$, $c_2 \in \mathbb{Z} \cup \{+\infty\}$ range over mathematical integer values possibly extended to include (negative or positive) infinity, $\diamond \in \{+, -, *, \dots\}$, and $\bowtie \in \{=, \neq, \leq, \dots\}$. A unique label $l \in \mathcal{L}$ appears within each instruction statement. Variable assignments can involve an externally-controlled input ($^l x := \mathsf{input}$), a non-deterministic choice ($^l x := [c_1, c_2]$) or an arithmetic expression ($^l x := \mathtt{a}$). A program is an instruction statement $\mathtt{s} \in \mathrm{Stmt}$ followed by a label $l \in \mathcal{L}$.

Figure 4 shows two programs written in our small language, both slices of program P from Figure 2. Program P_1 (cf. Figure 4a) is akin to the transition system in Example 1: the input $x \leq 0$ leads to non-termination (state c in Figure 3a), while for $x > 0$ (state b) the program always terminates. Instead, P_2 (cf. Figure 4b) is akin to Example 2: for the input $x \leq 10$ (state b in Figure 3b), termination is possible with $c < 0$ (state d).

In this section, we instantiate the definition of our termination resilience semantics Λ_{tr} with respect to programs $\mathtt{p} \in \mathrm{Prog}$ written in our small language. Then, in the next section, we will define its abstraction $\Lambda^{\natural}_{\mathsf{tr}}$.

Taint Semantics. An environment $\mathcal{E} \colon \mathbb{X} \to \mathbb{Z}$ maps program variables to their value. The semantics $[\![\mathtt{e}]\!] \colon \mathcal{E} \rightharpoonup \mathcal{P}(\mathbb{Z})$ of an expression $\mathtt{e} \in \mathrm{AExp} \cup \{\mathsf{input}, [c_1, c_2]\}$ maps an environment to the possible values of the expression $\mathtt{e}$. Its definition is standard and, for the sake of completeness, is given in Appendix A.

Program states are pairs $\mathcal{L} \times \mathcal{E}$ of a label $l \in \mathcal{L}$ and an environment $\rho \in \mathcal{E}$ defining the values of the program variables at the program point designated by l. To keep track of the effect of input transitions — i.e., variable assignments with externally-controlled inputs ($^l x := \mathsf{input}$) — on program states, we define in Figure 5 a *taint semantics* $\mathfrak{T}[\![\mathtt{s}]\!] : \mathcal{P}(\mathbb{X}) \to \mathcal{P}(\mathbb{X})$ that collects the set of variables that depend on external inputs, i.e., are *tainted*, after the execution of a program statement. Variable assignments add the assigned variable to the set of tainted variables if the assigning expression is an externally-controlled input ($^l x := \mathsf{input}$) or ($^l x := \mathtt{a}$) if its value depends on already tainted variables.

We define the taint semantics $\mathfrak{T}[\![\mathtt{a}]\!] \colon \mathcal{P}(\mathbb{X}) \to \{\mathsf{true}, \mathsf{false}\}$ of expressions as:

$$\mathfrak{T}[\![\mathtt{a}]\!]\mathrm{T} \triangleq \exists \rho \in \mathcal{E} \colon \exists V \in \mathbb{Z}^{|\mathbb{X}|} \colon V \in [\![\mathtt{a}]\!]\rho \wedge \forall \rho' \neq_{\mathrm{T}} \rho \colon V \notin [\![\mathtt{a}]\!]\rho' \tag{8}$$

$$
\begin{aligned}
\mathfrak{T}[\![^l \text{ skip}]\!]\mathrm{T} &\triangleq \mathrm{T} \\
\mathfrak{T}[\![^l x := \mathsf{input}]\!]\mathrm{T} &\triangleq \mathrm{T} \cup \{x\} \\
\mathfrak{T}[\![^l x := [c_1, c_2]]\!]\mathrm{T} &\triangleq \mathrm{T} \setminus \{x\} \\
\mathfrak{T}[\![^l x := \mathtt{a}]\!]\mathrm{T} &\triangleq \begin{cases} \mathrm{T} \cup \{x\} & \mathfrak{T}[\![\mathtt{a}]\!]\mathrm{T} \\ \mathrm{T} \setminus \{x\} & \text{otherwise} \end{cases} \\
\mathfrak{T}[\![\text{if}^l \mathtt{a} \bowtie 0 \{\mathtt{s}\}]\!]\mathrm{T} &\triangleq \begin{cases} \mathfrak{T}[\![\mathtt{s}]\!]\mathrm{T} \cup \textsc{assigned}(\mathtt{s}) \cup \mathrm{T} & \mathfrak{T}[\![\mathtt{a}]\!]\mathrm{T} \\ \mathfrak{T}[\![\mathtt{s}]\!]\mathrm{T} \cup \mathrm{T} & \text{otherwise} \end{cases} \\
\mathfrak{T}[\![\text{while}^l \mathtt{a} \bowtie 0 \{\mathtt{s}\}]\!]\mathrm{T} &\triangleq \mathsf{lfp}^{\subseteq}_{\mathrm{T}} \mathfrak{T}[\![\text{if}^l \mathtt{a} \bowtie 0 \{\mathtt{s}\}]\!] \\
\mathfrak{T}[\![\mathtt{s}_1; \mathtt{s}_2]\!]\mathrm{T} &\triangleq \mathfrak{T}[\![\mathtt{s}_2]\!](\mathfrak{T}[\![\mathtt{s}_1]\!]\mathrm{T})
\end{aligned}
$$

Fig. 5: Taint semantics $\mathfrak{T}[\![\mathtt{s}]\!] : \mathcal{P}(\mathbb{X}) \to \mathcal{P}(\mathbb{X})$, where $\textsc{assigned}(\mathtt{s})$ is the set of variables assigned with an arithmetic expression within the statement $\mathtt{s}$.

where $\rho' \neq_{\mathrm{T}} \rho \Leftrightarrow \exists \emptyset \subset \mathrm{T}' \subseteq \mathrm{T} : (\forall x \notin \mathrm{T}' : \rho(x) = \rho'(x)) \wedge (\forall x \in \mathrm{T}' : \rho(x) \neq \rho'(x))$ denotes program environments that differ only on (a non-empty subset of) the tainted variables in T. An expression is tainted if there exists a value of its semantics $[\![\mathtt{a}]\!]$ that is only possible in an environment ρ with a certain value V for the set T of tainted variables, i.e., the value of the expression depends on externally-controlled program inputs. Note that this definition differs from a classical non-interference-like definition (e.g., $\mathfrak{T}[\![\mathtt{a}]\!]\mathrm{T} \triangleq \forall \rho, \rho' \in \mathcal{E} : \forall x \notin \mathrm{T} : \rho(x) = \rho'(x) \wedge [\![\mathtt{a}]\!]\rho \neq [\![\mathtt{a}]\!]\rho'$) to also take into account the non-determinism arising from uncontrolled variables (i.e., $[\![\mathtt{a}]\!]\rho \neq [\![\mathtt{a}]\!]\rho'$ could be the consequence of non-deterministic value choices rather than externally-controlled inputs). In Appendix B, we propose a more precise taint semantics that takes into account the set of reachable environments at each program label.

The taint semantics of compound statements (if and while), if their guard $\mathtt{a} \bowtie 0$ is tainted, also adds to the tainted variables the set $\textsc{assigned}(\mathtt{s})$ of variables assigned with arithmetic expression $(^l x := \mathtt{a})$ in their body $\mathtt{s}$. The taint semantics of while statements is the least fixpoint of the taint semantics of if statements: $\lambda X. \mathfrak{T}[\![\text{if}^l \mathtt{a} \bowtie 0 \{\mathtt{s}\}]\!]X$ is monotone and $\langle \mathcal{P}(\mathbb{X}), \subseteq \rangle$ is a finite ordered set, so the fixpoint exists. The taint semantics $\mathfrak{T}[\![\mathtt{p}]\!] \in \mathcal{P}(\mathbb{X})$ of a program $\mathtt{p} \in \text{Prog}$ is then $\mathfrak{T}[\![\mathtt{p}]\!] = \mathfrak{T}[\![\mathtt{s}^l]\!] \triangleq \mathfrak{T}[\![\mathtt{s}]\!]\emptyset$. By pointwise-lifting of $\mathfrak{T}$ with respect to the program labels $\mathcal{L}$, we obtain the lifted taint semantics $\dot{\mathfrak{T}} : \mathcal{L} \to \mathcal{P}(\mathbb{X})$ mapping each label to the set of variables that are taint at that label.

Example 7. Let us consider again the example programs in Figure 4. The set $\dot{\mathfrak{T}}(2)$ of taint variables at label 2 is $\{x\}$ for both programs.

For P_1, the fixpoint iterations at label 3 add z because at label 4 it is assigned with an arithmetic expression that depends on already tainted variables ($\mathfrak{T}[\![\mathtt{z} - \mathtt{x}]\!]\{x\}$ is true); the fixpoint is thus $\{x, z\}$.

$$\Lambda_{\mathsf{tr}}[\![{}^{l}\,\mathtt{skip}]\!]f \triangleq \lambda\rho \in dom(f).f(p)+1$$

$$\Lambda_{\mathsf{tr}}[\![{}^{l}\,x := \mathsf{e}]\!]f \triangleq \lambda\rho. \begin{cases} \sup\{f(\rho[x \leftarrow v]) + 1 \mid v \in [\![\mathsf{e}]\!]\rho\} & \mathsf{e} = \mathsf{input} \vee A \\ \inf\left\{ f(\rho[x \leftarrow v]) + 1 \;\middle|\; \begin{array}{c} v \in [\![\mathsf{e}]\!]\rho \wedge \\ \rho[x \leftarrow v] \in \mathsf{dom}(f) \end{array} \right\} & \mathsf{e} = [c_1, c_2] \vee B \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$A \Leftrightarrow (\mathsf{e} = \mathsf{a} \wedge \mathfrak{T}[\![\mathsf{a}]\!](\dot{\mathfrak{T}}(l)) \wedge \exists v \in [\![\mathsf{a}]\!]\rho \wedge \forall v \in [\![\mathsf{a}]\!]\rho, \rho[x \leftarrow v] \in \mathsf{dom}(f))$$

$$B \Leftrightarrow (\mathsf{e} = \mathsf{a} \wedge \neg\mathfrak{T}[\![\mathsf{a}]\!](\dot{\mathfrak{T}}(l)) \wedge \exists v \in [\![\mathsf{a}]\!]\rho : \rho[x \leftarrow v] \in \mathsf{dom}(f))$$

$$\Lambda_{\mathsf{tr}}[\![\mathtt{if}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathtt{s}\}]\!]f \triangleq \Theta_{\mathsf{tr}}(f)$$

$$\Theta_{\mathsf{tr}} \triangleq \lambda X.\lambda\rho. \begin{cases} \sup\{\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]X(\rho), f(\rho)+1\} & C \\ \inf\{\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]X(\rho), f(\rho)+1\} & D \\ \Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]X(\rho)+1 & E \vee G \\ f(\rho)+1 & F \vee H \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\textsc{non-det} \Leftrightarrow \exists v_1, v_2 \in [\![\mathsf{a}]\!]\rho : v_1 \bowtie 0 \wedge v_2 \not\bowtie 0$$

$$C \Leftrightarrow \textsc{non-det} \wedge \mathfrak{T}[\![\mathsf{a}]\!]\dot{\mathfrak{T}}(l) \wedge \rho \in \mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]f) \cap \mathsf{dom}(f)$$

$$D \Leftrightarrow \textsc{non-det} \wedge \neg\mathfrak{T}[\![\mathsf{a}]\!]\dot{\mathfrak{T}}(l) \wedge \rho \in \mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]f) \cap \mathsf{dom}(f)$$

$$E \Leftrightarrow \textsc{non-det} \wedge \neg\mathfrak{T}[\![\mathsf{a}]\!]\dot{\mathfrak{T}}(l) \wedge \rho \in \mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]f) \setminus \mathsf{dom}(f)$$

$$F \Leftrightarrow \textsc{non-det} \wedge \neg\mathfrak{T}[\![\mathsf{a}]\!]\dot{\mathfrak{T}}(l) \wedge \rho \in \mathsf{dom}(f) \setminus \mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]f)$$

$$G \Leftrightarrow \forall v \in [\![\mathsf{a}]\!]\rho : v \bowtie 0 \wedge \rho \in \mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]f(\rho))$$

$$H \Leftrightarrow \forall v \in [\![\mathsf{a}]\!]\rho : v \not\bowtie 0 \wedge \rho \in \mathsf{dom}(f(\rho))$$

$$\Lambda_{\mathsf{tr}}[\![\mathtt{while}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathtt{s}\}]\!]f \triangleq \mathsf{lfp}^{\sqsubseteq}_{\dot{\emptyset}}\,\Theta_{\mathsf{tr}}$$

$$\Lambda_{\mathsf{tr}}[\![\mathtt{s}_1;\mathtt{s}_2]\!]f \triangleq \Lambda_{\mathsf{tr}}[\![\mathtt{s}_1]\!](\Lambda_{\mathsf{tr}}[\![\mathtt{s}_2]\!]f)$$

Fig. 6: Termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$.

For P_2, the iterations at label 4 also add z but, this time, because the boolean condition of the loop depends on tainted variables ($\mathfrak{T}[\![\mathtt{z} \geq \mathtt{x}]\!]\{x\}$ is true) and z belongs to $\textsc{assigned}({}^{5}z := z + c)$; the fixpoint is thus again $\{x, z\}$. ◁

Denotational Termination Resilience Semantics. We can now define in Figure 6 the termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ for each program statement. Each transfer function $\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]$ takes as input a (potential) ranking function that is valid *after* the execution of the statement $\mathtt{s}$ and returns a (potential) ranking function that is valid *before* the execution of $\mathtt{s}$.

If the value on the right-hand side of a variable assignment is an externally-controlled input (${}^{l}\,x := \mathsf{input}$) or an expression a (${}^{l}\,x := \mathsf{a}$) that depends on tainted variables ($\mathfrak{T}[\![\mathsf{a}]\!](\dot{\mathfrak{T}}(l))$), the termination resilience $\Lambda_{\mathsf{tr}}[\![{}^{l}\,x := \mathsf{a}]\!]$ returns a (potential) ranking function only defined over the environments ρ that when

subject to the variable assignment ($\rho[x \leftarrow v]$ with $v \in [\![\mathsf{a}]\!]\rho$) *always* belong to the domain of the given (potential) ranking function f (case A). The value of f for these environments is increased by one, to take into account another program execution step before termination, and the value of the resulting ranking function is the least upper bound of these values (maximum number of steps to termination). Otherwise – if the value on the right-hand side of the assignment is a non-deterministic choice (${}^l\, x := [c_1, c_2]$) or an expression a that does not depend on tainted variables ($\neg\mathfrak{T}[\![\mathsf{a}]\!](\dot{\mathfrak{T}}(l))$) – the resulting function is defined over the environments ρ for which *at least one* value $v \in [\![\mathsf{a}]\!]\rho$ yields an environment $\rho[x \leftarrow v]$ in $\mathsf{dom}(f)$ (case B). Its value is the greatest lower bound of the value of f on these environments plus one (minimum number of steps to termination).

Example 8. Let us consider again the program P_2 in Figure 4b. Let

$$f \triangleq \lambda\rho. \begin{cases} 1 & \rho(z) < \rho(x) \\ \frac{2\rho(z) - 2\rho(x)}{-\rho(c)} + 3 & \rho(z) \geq \rho(x) \wedge \rho(c) < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

be the fixpoint at program label 4 and let us consider the non-deterministic variable assignment ${}^3c := [-2, 1]$. We have $[\![[-2, 1]]\!]\rho = \{-2, -1, 0, 1\}$ for any $\rho \in \mathcal{E}$ but, if $z \geq x$, $\rho[c \leftarrow v] \in \mathsf{dom}(f)$ only when $v \in [\![[-2, 1]]\!]\rho$ is negative, i.e., when $v = -1$ or $v = -2$. Thus, the termination resilience semantics at label 3 is

$$\lambda\rho. \begin{cases} 2 & \rho(z) < \rho(x) \\ \rho(z) - \rho(x) + 4 & \text{otherwise} \end{cases}$$

since $\inf\{\frac{2\rho(z) - 2\rho(x)}{1} + 4, \frac{2\rho(z) - 2\rho(x)}{2} + 4\} = \rho(z) - \rho(x) + 3$. ◁

The termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\text{if}^{\,l}\, \mathsf{a} \bowtie 0\, \{\mathtt{s}\}]\!]$ for `if` statements, via Θ_{tr}, defines a potential ranking function distinguishing over environments that, due to non-determinism, *may* both satisfy and not satisfy the guard $\mathsf{a} \bowtie 0$ (cf. NON-DET), and environments that *must* satisfy (resp. not satisfy) it (case G, resp. case H). Additionally, it accounts for whether a depends on tainted variables (case C) or not (cases D, E, F). In the latter case, the potential ranking function is defined on all environments in $\mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]f)$ or $\mathsf{dom}(f)$, i.e., all environments potentially leading to a terminating execution. The termination resilience $\Lambda_{\mathsf{tr}}[\![\text{while}^{\,l}\, \mathsf{a} \bowtie 0\, \{\mathtt{s}\}]\!]$ for `while` statements is the least fixpoint of Θ_{tr} with respect to the computational order $f_1 \sqsubseteq f_2$ (cf. Definition 3).

Example 9. Let us consider again the program P_1 in Figure 4a. The fixpoint iterations at label 3 start with the totally undefined function $f_3^0 \triangleq \dot{\emptyset}$. Let f be the constant function equal to zero $\lambda\rho.0$. The first fixpoint iteration yields

$$f_3^1 \triangleq \lambda\rho. \begin{cases} 1 & \rho(z) < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

via case H of Θ_{tr}. The assignment at label 4 yields the function

$$\lambda\rho. \begin{cases} 2 & \rho(z) - \rho(x) < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the second fixpoint iteration thus yields

$$f_3^2 \triangleq \lambda\rho. \begin{cases} 1 & \rho(z) < 0 \\ 3 & 0 \leq \rho(z) < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

via case H of Θ_{tr} for $\rho(z) < 0$ and case G for $0 \leq \rho(z) < \rho(x)$. Finally,

$$f_3 \triangleq \lambda\rho. \begin{cases} 1 & \rho(z) < 0 \\ \frac{2\rho(z)}{\rho(x)} + 3 & 0 \leq \rho(z) \wedge 0 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

is the least fixpoint at program label 3. ◁

The termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\mathtt{p}]\!] \in (\mathcal{E} \rightharpoonup \mathbb{O})$ of a program $\mathtt{p} \in \mathrm{Prog}$ is determined by $\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]$ taking as input the constant function equal to zero.

Definition 4 (Denotational Termination Resilience Semantics). *Let* $\mathtt{p} \in \mathrm{Prog}$ *be a program. Its denotational termination resilience semantics is:*

$$\Lambda_{\mathsf{tr}}[\![\mathtt{p}]\!] = \Lambda_{\mathsf{tr}}[\![\mathtt{s}^l]\!] \triangleq \Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!](\lambda\rho.0). \tag{9}$$

Example 10. Let us consider again the programs P_1 and P_2 in Figure 4.

For program P_1, continuing Example 9, we obtain

$$\lambda\rho. \begin{cases} \frac{20}{\rho(x)} + 4 & 0 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

at program label 2 and, finally, $\dot{\emptyset}$ at label 1, i.e., $\Lambda_{\mathsf{tr}}[\![P_1]\!] \notin \mathcal{TR}$.

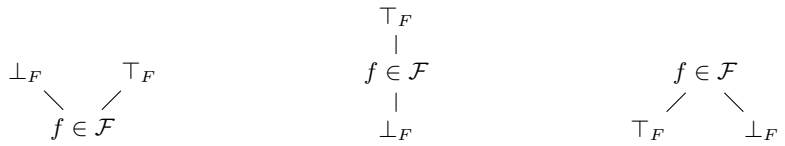
For program P_2, continuing Example 8, we get

$$\lambda\rho. \begin{cases} 3 & 10 < \rho(x) \\ 15 - \rho(x) & \text{otherwise} \end{cases}$$

at label 2 and, finally $\lambda\rho.\omega$ at label 1, i.e., $\Lambda_{\mathsf{tr}}[\![P_2]\!] \in \mathcal{TR}$. ◁

5 Abstract Termination Resilience Semantics

In this section, we propose a sound abstraction $\Lambda_{\mathsf{tr}}^{\natural}[\![\mathtt{p}]\!]$ of the denotational termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\mathtt{p}]\!]$ defined in Section 4 by leveraging (and extending) Urban and Miné's decision tree numerical abstract domain $\mathcal{T}$ [33] to



(a) Approximation order $\preceq_F$ (b) Computational order $\sqsubseteq_F$ (c) Resilience order $\leq_F$

Fig. 8: Hasse diagrams for difference orders of decision tree leaves.

abstract potential ranking functions by means of piecewise-defined partial functions. Specifically, we consider the concretization-based abstraction [11] $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \preceq \rangle \xleftarrow{\gamma} \langle \mathcal{T}, \preceq_T \rangle$ where the approximation order $\preceq$ is defined as follows:

$$f_1 \preceq f_2 \Leftrightarrow \mathsf{dom}(f_1) \supseteq \mathsf{dom}(f_2) \wedge \forall x \in \mathsf{dom}(f_2) \colon f_1(x) \leq f_2(x). \tag{10}$$

We define $\Lambda^{\natural}_{\mathsf{tr}}[\![\mathtt{p}]\!]$ to have $\Lambda_{\mathsf{tr}}[\![\mathtt{p}]\!] \preceq \gamma(\Lambda^{\natural}_{\mathsf{tr}}[\![\mathtt{p}]\!])$ meaning that $\Lambda^{\natural}_{\mathsf{tr}}[\![\mathtt{p}]\!]$ *over-approximates* the value of $\Lambda_{\mathsf{tr}}[\![\mathtt{p}]\!]$ and *under-approximates* its domain $\mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathtt{p}]\!])$. In this way, the abstraction $\Lambda^{\natural}_{\mathsf{tr}}[\![\mathtt{p}]\!]$ provides *sufficient preconditions* for termination resilience.

Decision Tree Abstract Domain. An element $t \in \mathcal{T}$ is a piecewise-defined partial function represented by a decision tree, where each node $\textsc{node}\{c\}\colon t_1, t_2$, $t_1, t_2 \in \mathcal{T}$, is labeled by a constraint $c \in \mathcal{C}$ over the program variables, and each leaf $\textsc{leaf}\colon f$ is labeled by a function $f \in \mathcal{F}$ of the program variables. Nodes recursively partition the space of possible values of the program variables: constraint labeling nodes are satisfied by the left subtrees of the nodes, while the right subtrees satisfy their negation. Leaves represent the value of the function corresponding to each partition. Figure 7 shows an example of decision tree. The leaf with value $\bot_F$ explicitly represents the undefined partition of the (partial) function. Undefined functions can also be represented by the leaf value $\top_F$ in case of an irrecoverable loss of precision of the analysis [8]. The decision tree in Figure 7 is automatically inferred by our termination resilience static analysis at label 4 of program P_2 in Figure 4b. It represents a function with constant value 1 when $z < x$ (at most one step to termination), with constant value 3 when $z + c < x$ (at most three steps to termination) and undefined otherwise (termination is not proved when $z + c \geq x$), while the concrete termination resilience semantics at the same label 4 is defined when $z < x$ or $c < 0$ (cf. Example 8): the decision tree is thus a sound approximation with respect to $\preceq$ (cf. Equation 10).

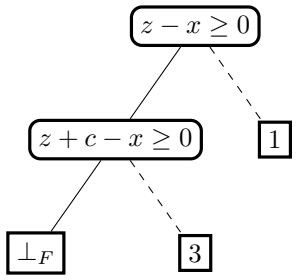


Fig. 7: Decision tree inferred by our analysis at label 4 of program P_2 (cf. Figure 4b).

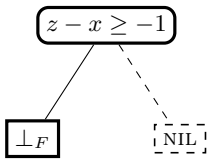
Algorithm 1 Decision Tree Assignment

```
1:  function ASSIGN_T⟦^l x := e⟧(t)
2:      if isLeaf(t) then
3:          return LEAF: ASSIGN_F⟦^l x := e⟧(t.f)
4:      else if isNode(t) then
5:          I ← ASSIGN_C⟦^l x := e⟧(t.c)
6:          J ← ASSIGN_C⟦^l x := e⟧(¬t.c)
7:          if isEmpty(I) ∧ isEmpty(J) then
8:              return ASSIGN_T⟦^l x := e⟧(t.l) ϒ_T ASSIGN_T⟦^l x := e⟧(t.r)
9:          else if isEmpty(I) ∧ ⊥_C ∈ J then
10:             return ASSIGN_T⟦^l x := e⟧(t.l)
11:         else if ⊥_C ∈ I ∧ isEmpty(J) then
12:             return ASSIGN_T⟦^l x := e⟧(t.r)
13:         else
14:             l ← ASSIGN_T⟦^l x := e⟧(t.l)
15:             t_1 ← PRUNE_T(l, I)
16:             r ← ASSIGN_T⟦^l x := e⟧(t.r)
17:             t_2 ← PRUNE_T(r, J)
18:             return t_1 ϒ_T t_2
```

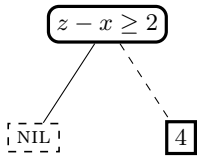
The partitioning induced by decision tree constraints is dynamic: the analysis begins at the end of the program with the decision tree LEAF: 0 (i.e., zero program executions steps to termination) and proceeds *backwards*; during the analysis, constraints are added by boolean conditions or when merging control flows, and are modified by variable assignments.

Algorithm 1 shows the decision tree assignment operator $\text{ASSIGN}_T\llbracket^l x := \mathsf{e}\rrbracket$ originally defined in [31]. The assignment is performed by recursively descending the given decision tree t (cf. Lines 14 and 16, the left subtree is accessed with $t.l$ and the right one with $t.r$). At the leaves, the assignment handled by the auxiliary operator $\text{ASSIGN}_F\llbracket^l x := \mathsf{e}\rrbracket$ operating on functions (accessed by $t.f$, cf. Line 3): after the assignment, the value of defined functions is increased by one to account for one more step to termination. For example, let us consider the decision tree in Figure 7 and the assignment $c := [-2, 1]$ at label 3 in Figure 4b. The result of the assignment on LEAF : 3 is LEAF : 4, simply increasing the value of its function by one, while LEAF : $\bot_F$ remains unchanged. Node constraints (accessed with $t.c$) and their negation ($\neg t.c$) are handled by the auxiliary operator $\text{ASSIGN}_C\llbracket^l x := \mathsf{e}\rrbracket$ of the underlying numerical abstract domain $\mathcal{C}$ used to manage constraints, which produces a set $I \in \mathcal{P}(\mathcal{C})$ (cf. Line 5) and a set $J \in \mathcal{P}(\mathcal{C})$ (cf. Line 6) of linear constraints resulting from the assignment to $t.c$ and $\neg t.c$, respectively. For example, considering again the decision tree in Figure 7, the result of the assignment $c := [-2, 1]$ on the constraint $z+c-x \geq 0$ and its negation $z+c-x < 0$ yields $I = \{z - x \geq -1\}$ and $J = \{z - x < 2\}$. The set of constraints I and J are used to prune the subtrees resulting from the recursive calls (cf. Lines 15 and 17). In our example, the pruning of the LEAF : $\bot_F$ with the set of constraints I results in the decision tree shown in Figure 9a), while the pruning of LEAF : 4

with the set J results in the tree in Figure. 9b.



(a) Pruning of LEAF: $\bot_F$ with $I = \{z - x \geq -1\}$.



(b) Pruning of LEAF: 4 with $J = \{z - x < 2\}$.

Fig. 9: Decision trees resulting from PRUNE_T.

The special NIL node denotes partitions that are outside the domain of definition of the decision tree. Note that it differs from leaves labeled with $\bot_F$ or $\top_F$, which denote undefined partitions within the domain of definition of the tree. Finally, the resulting trees are joined by the approximation join $\curlyvee_T$ (cf. Line 18), which merges the domains of definitions of the joined trees (essentially removing any NIL nodes) and retains the leaves labeled with an undefined function ($\bot_F$ or $\top_F$) in either of the joined decision trees, cf. the Hasse diagram in Figure 8a. Continuing the example, joining the trees in Figure 9 yields the tree in Figure 10. In case both I and J are empty (cf. Line 7), neither the constraint $t.c$ nor its negation $\neg t.c$ exists anymore and thus the subtrees resulting from the recursive calls are joined by the approximation join $\curlyvee_T$. In case I is empty and J is an unsatisfiable set of constraints (i.e., the unsatisfiable constraint $\bot_C$ belongs to J, cf. Line 9), it means that $\neg t.c$ is no longer satisfiable and thus only the left subtree of the decision tree is kept (cf. Line 10). Similarly, in case I is an unsatisfiable set of constraints and J is empty (cf. Line 11), only the right subtree of the decision tree is kept (cf. cf. Line 12).

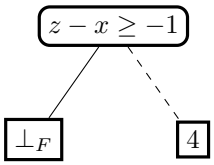


Fig. 10: Approximation join of Figure 9a and Figure 9b.

To minimize the cost of the analysis and to enforce its termination, a widening operator $\triangledown_T$ limits the height of the decision trees and the number of induced partitions. Abstract fixpoint iterations with widening follow the computational order $\sqsubseteq_T$, which preserves decision tree leaves labeled with a defined function over undefined leaves labeled with $\bot_F$ (i.e., the analysis grows the domain of the inferred raking function at each iteration), but preserves leaves labeled with $\top_F$ over all other leaves (i.e., the domain of the inferred function shrinks when the analysis loses too much precision), cf. Figure 8b.

Non-Deterministic Assignments. To leverage the decision tree abstract domain for termination resilience we need to extend it to distinguish between variables assigned with externally controlled or non-deterministic values. Termination must be ensured for *all* values of variables that are externally controlled,

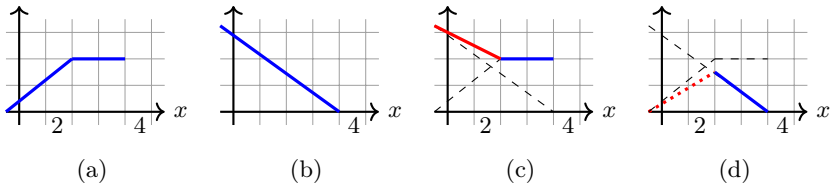


Fig. 11: Join of two partial piecewise-defined functions (a)(b) using the approximation $\curlyvee_T$ or computational $\sqcup_T$ join (c) and the resilience $\veebar_T$ join (d). The result of the resilience join is undefined for $x \leq 2$.

while for non-deterministic assignments, *any* value that ensures termination is enough to satisfy termination resilience. We thus introduce a new *resilience join* operator $\veebar_T$ between decision trees that retains the leaves labeled with a *defined* function in either of the joined decision trees, cf. the Hasse diagram in Figure 8c.

The join of two leaves labeled with defined functions $f_1, f_2 \in \mathcal{F}$ must over-approximate the value of the concrete ranking function (cf. Equation 10). To over-approximate the number of steps to termination, the approximation and computational join preserve the leaf labeled with the function with the largest value, or find a third function $f \in \mathcal{F}$ with value larger than both (returning $\top_F$ if this cannot be found within $\mathcal{F}$), cf. Figure 11c for an example. Instead, the resilience join, preserves the leaf with the function with the *smallest* value, returning $\top_F$ if the functions are not comparable, cf. Figure 11d. Note that, the resilience join would not be sound if it returned a third function $f \in \mathcal{F}$ with value smaller than both f_1 and f_2 (cf. the red dotted line in Figure 11d) since this may under-approximate the value of the concrete ranking function.

We implement the operator $\textsc{assign}_T[\![^l\, x := [c_1, c_2]]\!]$ for non-deterministic assignments following Algorithm 1, but using the resilience join $\veebar_T$ instead of the approximation join $\curlyvee_T$. Specifically, Line 8 of Algorithm 1 is changed to:

return $\textsc{assign}_T[\![^l\, x := \mathsf{e}]\!](t.l) \veebar_T \textsc{assign}_T[\![^l\, x := \mathsf{e}]\!](t.r)$

and Line 18 is changed as follows:

return $l \veebar_T r$

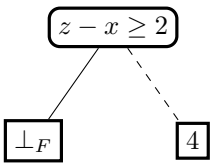


Fig. 12: Resilience join of Figure 9a and Figure 9b.

Let us consider again the left subtree in Figure 7 and the non-deterministic variable assignment $c := [-2, 1]$ at label 3 in Figure 4b. Joining the trees in Figure 9 resulting from the $\textsc{prune}_T$ at Lines 15 and 17 yields the decision tree in Figure 12, which indicates that the termination of program P_2 is possible when $z - x < 2$, for at least *some* values of c chosen at label 3 (notably, for $c \in \{-2, -1\}$, but the analysis does not explicitly provide these values). The final decision tree at program label 3 is shown in Figure 13.

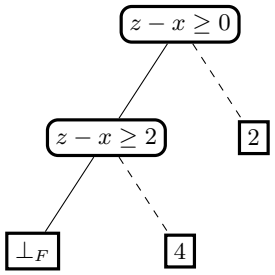


Fig. 13: Decision tree inferred by our analysis at label 3 of program P_2.

Note that this seemingly anodyne modification is actually insidious. Let us consider the decision tree in Figure 14 and the variable assignment $x := [-\infty, +\infty]$, and assume that the variable y is externally-controlled. The result of the assignment on the decision tree constraints is the empty set of constraints and, since the variable x is assigned with a non-deterministic value, we can employ the resilience join $\veebar_F$ to join all leaves after the assignment is performed on them. This yields the final decision tree LEAF: 2 (obtained from LEAF: 1 after the assignment), which is not sound! In fact, the sufficient precondition for termination resilience given by the decision tree in Figure 14 requires $y > 0$ but this constraint is missing from the decision tree (because redundant). As a consequence, when the assignment is performed and the other constraints $x < 0$ and $x + y \geq 0$ disappear, the result is an unsound decision tree indicating that termination resilience always holds. To avoid this issue, before non-deterministic variable assignments, decision trees need to be *filled* to explicitly contain redundant constraints. Specifically, we further modify Algorithm 1 to insert the following *before* Line 2:

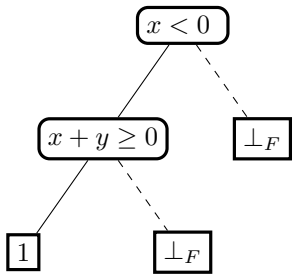


Fig. 14: Unfilled decision tree.

if $\mathsf{e} = [c_1, c_2]$ **then**
  $\textsc{fill}_T(t)$

where $\textsc{fill}_T$ leverages the underlying numerical abstract domain $\mathcal{C}$ to find these redundant constraints and adds them to the decision tree.

The $\textsc{assign}_T$ operator for non-deterministic variable assignments implemented by Algorithm 1 modified as discussed above is a sound over-approximation of the termination resilience semantics (cf. Figure 6):

Proposition 1. $\Lambda_{\mathsf{tr}}[\![^l\, x := [c_1, c_2]]\!]\gamma(t) \preceq \gamma(\textsc{assign}_T[\![^l\, x := [c_1, c_2]]\!]t)$

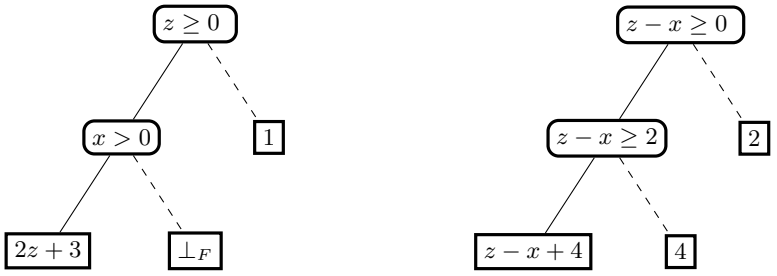
where the concretization function $\gamma \colon \mathcal{T} \to (\mathcal{E} \rightharpoonup \mathbb{O})$ maps a decision tree to the represented piecewise-defined partial (potential ranking) function.

Proof (Sketch). Let $C \triangleq \Lambda_{\mathsf{tr}}[\![^l\, x := [c_1, c_2]]\!]\gamma(t)$ and let $A \triangleq \gamma(\textsc{assign}_T[\![^l\, x := [c_1, c_2]]\!]t)$. We prove that $\mathsf{dom}(A) \subseteq \mathsf{dom}(C)$ and $\forall \rho \in \mathsf{dom}(A)\colon C(\rho) \leq A(\rho)$.

Let us assume by absurd that $\mathsf{dom}(C) \subset \mathsf{dom}(A)$. Then, there exists $\rho \in \mathcal{E}$ such that $\rho \in \mathsf{dom}(A)$ and $\rho \notin \mathsf{dom}(C)$. Since $\rho \in \mathsf{dom}(A)$ then, by definition of $\textsc{assign}_T[\![^l\, x := [c_1, c_2]]\!]$ (i.e., Algorithm 1 modified as discussed above), we have $\exists v \in [\![[c_1, c_2]]\!]\rho\colon \rho[x \leftarrow v] \in \mathsf{dom}(\gamma(t))$. Thus, by definition of $\Lambda_{\mathsf{tr}}[\![^l\, x := [c_1, c_2]]\!]$ (cf. Figure 6), $\rho \in \mathsf{dom}(C)$, which is absurd. Therefore, $\mathsf{dom}(A) \subseteq \mathsf{dom}(C)$.

Let us now assume by absurd that $\exists \rho \in \mathsf{dom}(A)\colon C(\rho) > A(\rho)$. By definition of $\Lambda_{\mathsf{tr}}[\![^l\, x := [c_1, c_2]]\!]$, we have $C(\rho) = \inf \left\{ \gamma(t)(\rho[x \leftarrow v]) + 1 \,\middle|\, v \in [\![\mathsf{e}]\!]\rho \right\}$. Moreover, by the definition of $\textsc{assign}_T[\![^l\, x := [c_1, c_2]]\!]$ and, notably, the definition of the resilience join $\veebar_F$ (cf. Figure 11), we have $\inf \left\{ \gamma(t)(\rho[x \leftarrow v]) + 1 \,\middle|\, v \in [\![\mathsf{e}]\!]\rho \right\} \leq A(\rho)$ since $\rho \in \mathsf{dom}(A)$. Thus, $C(\rho) \leq A(\rho)$, which is absurd. Therefore, we have $\forall \rho \in \mathsf{dom}(A)\colon C(\rho) \leq A(\rho)$ and this concludes the proof. □

Resilience Join and Non-Tainted Program Variables. At this point, one may be tempted to extend the use of the resilience join to all variable assignment with a non-tainted right-hand side expression (i.e., an expression that does not depend from externally controlled variables) or, similarly, to `if` and `while` statements with non-tainted boolean conditions. However, this change alone would be unsound. Consider program P in Figure 2 and let the boolean condition at label 3 be $a * a \geq 0$ where a is assigned with a non-deterministic value before x. Figure 15 shows the most precise decision trees that can be inferred at label 4 and 6.



(a) Decision tree at label 4. (b) Decision tree at label 6.

Fig. 15: Abstract termination resilience semantics for program P.

At program label 3, the boolean condition $a * a \geq 0$ cannot be precisely represented by a (set of) linear constraints. Thus, no further constrains are added before joining the trees. Since the decision tree at label 6 is totally defined (cf. Figure 15b), using the resilience join would also yield a decision tree that is totally defined (cf. Figure 8c), implying that termination resilience always holds. However, program label 6 is not reachable and, indeed, termination resilience does not hold: the program is robustly non-terminating when $x \leq 0$.

To ensure soundness in the general case of non-tainted variables and expressions, the resilience join should retain leaves labeled with a defined function only when the domain partition that they represent is actually reachable. In practice,

$$\Lambda^\natural_{\mathsf{tr}}[\![^l \,\mathtt{skip}]\!]t \triangleq \textsc{skip}_T(t)$$
$$\Lambda^\natural_{\mathsf{tr}}[\![^l \, x := \mathtt{e}]\!]t \triangleq \textsc{assign}_T[\![^l \, x := \mathtt{e}]\!](t)$$
$$\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{if}^{\,l}\, \mathtt{a} \bowtie 0 \,\{\mathtt{s}\}]\!]t \triangleq \Theta^\natural_{\mathsf{tr}}(t)$$
$$\Theta^\natural_{\mathsf{tr}} \triangleq \lambda X.\textsc{filter}_T[\![\mathtt{a} \bowtie 0]\!](\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}]\!]X) \curlyvee_T \textsc{filter}_T[\![\mathtt{a} \not\bowtie 0]\!](t)$$
$$\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{while}^{\,l}\, \mathtt{a} \bowtie 0 \,\{\mathtt{s}\}]\!]t \triangleq \textsc{lfp}^\natural \, \Theta^\natural_{\mathsf{tr}}$$
$$\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}_1; \mathtt{s}_2]\!]t \triangleq \Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}_2]\!](\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}_1]\!]t)$$

Fig. 16: Abstract Program Semantics for Termination Resilience

we ensure this by conservatively under-approximating the weakest liberal precondition of program statements by means of the approximation join [31]. We leave finding less conservative approximations for future work.

Abstract Termination Resilience Semantics. We can finally define in Figure 16 the *abstract termination resilience semantics* $\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}]\!] : \mathcal{T} \to \mathcal{T}$ for each program statement $\mathtt{s}$. The $\textsc{skip}_T$ operator simply increases by one the value of the functions labeling the leaves of the decision tree. The assignment operator $\textsc{assign}_T[\![^l \, x := \mathtt{e}]\!]$ is implemented by Algorithm 1, modified as discussed above for non-deterministic variable assignments. The semantics of conditional `if` statements $\Theta^\natural_{\mathsf{tr}}$ employs the approximation join $\curlyvee_T$ as discussed above, after handling the boolean conditions with the $\textsc{filter}_T$ operator. The semantics for `while` loops iterates $\Theta^\natural_{\mathsf{tr}}$ with widening starting from the totally undefined decision tree $\textsc{leaf} : \bot_F$ until an abstract fixpoint is reached.

The *abstract termination resilience semantics* $\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{p}]\!] \in \mathcal{T}$ for a $\mathtt{p} \in \mathrm{Prog}$ starts the termination resilience analysis with a decision tree containing a single leaf labeled with the zero function:

Definition 5 (Abstract Termination Resilience Semantics). *Let* $\mathtt{p} \in \mathit{Prog}$ *be a program. Its abstract termination resilience semantics is:*

$$\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{p}]\!] = \Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}^l]\!] \triangleq \Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}]\!](\mathit{LEAF} : 0). \tag{11}$$

The termination resilience analysis is sound with respect to the approximation order $\preceq$ (cf. Equation 10):

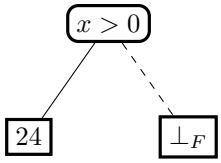
Theorem 2 (Soundness). $\Lambda_{\mathsf{tr}}[\![P]\!] \preceq \gamma(\Lambda^\natural_{\mathsf{tr}}[\![P]\!])$

where the concretization function $\gamma \colon \mathcal{T} \to (\mathcal{E} \rightharpoonup \mathbb{O})$ maps a decision tree to the represented piecewise-defined partial (potential ranking) function.

Proof (Sketch). The proof follows from Proposition 1 for $\Lambda^\natural_{\mathsf{tr}}[\![^l \, x := [c_1, c_2]]\!]$. For all other statements, we refer to the proofs in [31]. □

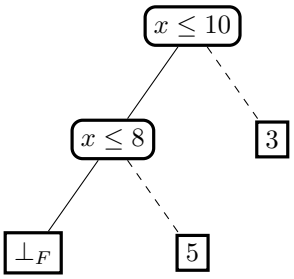
Corollary 1 (Soundness). $\Lambda[\![P]\!]I \in \mathcal{TR} \Leftarrow I \subseteq \mathsf{dom}(\gamma(\Lambda^\natural_{\mathsf{tr}}[\![P]\!]))$

Example 11. Let us consider again program P_1 in Figure 4a and its most precise termination resilience semantics at program label 3 shown in Figure 15a. The assignment at label 2 – substituting z with 10 and increasing by one the value of the functions labeling the defined leaves – yields



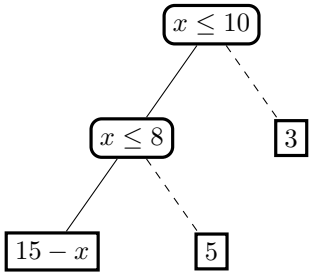
Finally, the assignment at label 1 yields $\textsc{leaf} : \bot_F$. Its concretization $\gamma(\textsc{leaf}: \bot_F)$ is the totally undefined function $\dot{\emptyset}$. We are thus unable to prove termination resilience. In this case, this is a true positive since $\Lambda_{\mathsf{tr}}[\![P_1]\!] \notin \mathcal{TR}$ (cf. Example 10). ◁

Example 12. Let us consider again program P_2 in Figure 4b. Its abstract termination resilience semantics at program label 3 is shown in Figure 13. The assignment at label 2 yields



and the assignment at label 1 yields $\textsc{leaf} : \bot_F$. We are again unable to prove termination resilience since $\gamma(\textsc{leaf}: \bot_F)$ is the totally undefined function $\dot{\emptyset}$. This, however, is a false alarm since $\Lambda_{\mathsf{tr}}[\![P_2]\!] \in \mathcal{TR}$ (cf. Example 10).

Let us consider instead the most precise abstract termination resilience semantics shown in Figure 15b. The assignment at label 2 yields



and the assignment at label 1 then yields $\textsc{leaf} : \omega$. In this case, we prove termination resilience since $\gamma(\textsc{leaf}: \omega)$ yields $\lambda\rho.\omega$, which is exactly $\Lambda_{\mathsf{tr}}[\![P_2]\!]$. ◁

6 Implementation and Experimental Evaluation

Implementation. We implemented our termination resilience analysis in the open-source tool LoopResilience[2]. LoopResilience is implemented in OCaml and relies on the Apron numerical abstract domain library [18] to manage decision tree constraints and functions labeling the tree leaves. It is also possible to activate the extension to lexicographic ranking functions [34], and tune the precision of the analysis by adjusting the widening delay. We use a simple implementation of the FILL_T operation (cf. Section 5), which adds constraints, obtained from Apron's interval domain, bounding the values of the variables assigned with a non-deterministic choice.

The analysis takes as input a numerical program written in a C-like syntax and outputs TRUE if all leaves of the inferred decision tree for the program are labelled with defined functions, i.e., Termination Resilience holds for any initial program state. Otherwise, it outputs UNKNOWN to indicate that Robust Non-Termination may hold. The defined partitions of the inferred decision tree are a sufficient precondition for Termination Resilience.

Non-terminating programs. We selected 64 non-terminating programs from three sources: SV-COMP 2024[3], the state-of-the-art non-termination analyzers Pulse [28] and a recent survey by Shi et al. on non-termination bugs [29]. Specifically, from SV-COMP categories *crafted-lit*, *restricted-15*, and *nla* we selected 40 programs labeled as non-terminating–i.e., the expected verdict for termination is false in the associated .yml file. From benchmarks of [28] (resp. from [29] benchmarks) we retained 11 (resp. 13) non-terminating programs. We excluded 16 (resp. 36) programs for which the property to prove is termination. Additionally, we discarded 34 (resp.42) programs manipulating pointers, bitwise operations or recursive functions (a non-trivial semantics work is required to handle these constructions).

Relevance of the selected programs. The selected programs from SV-COMP and Pulse are short programs (with an average length of 23 and 20 lines of code, respectively), which are typically used to evaluate termination and non-termination analyzers. To our knowledge, none of the available analyzers is capable of proving termination (or a property such as Termination Resilience) on real world programs. To evaluate our tool on more complex scenarios, we relied on the benchmark from Shi et al. [29]. They proposed a classification of non-termination bugs and, for each category, provided a corresponding set of small non-terminating programs (with an average of 25 lines of code). Each of these programs is a simplified version of a code extracted from real-world open-source software exhibiting a non-termination bug.

[2] Tool name and URL anonymized for review.
[3] https://sv-comp.sosy-lab.org/2024/

Experimental Evaluation. We evaluated LoopResilience on a benchmark of 235 test cases. Specifically, we constructed variants of the 64 non-terminating programs, for all possible combinations of variable initialization with externally-controlled (input) or non-deterministic ($[-\infty, +\infty]$) values. The experiments were conducted on a 64-bit 8-Core CPU (AMD® Ryzen 7 pro 5850u) with 16GB of RAM on Ubuntu 20.04.

Benchmark	Configuration	Property	Verified	Alarms	TO	Time
SV-COMP	LoopResilience-Boxes	Termination	0	119	0	3.5 s
		Termination Resilience	61	58	0	3.6 s
	LoopResilience-Polyhedra	Termination	0	119	0	7.2 s
		Termination Resilience	76	43	0	16.9 s
Pulse [28]	LoopResilience-Boxes	Termination	0	36	0	0.5 s
		Termination Resilience	16	20	0	0.5 s
	LoopResilience-Polyhedra	Termination	0	36	0	7.2 s
		Termination Resilience	16	20	0	16.9 s
Shi et al. [29]	LoopResilience-Boxes	Termination	0	85	0	2.0 s
		Termination Resilience	57	28	0	2.2 s
	LoopResilience-Polyhedra	Termination	0	85	0	69.0 s
		Termination Resilience	49	28	8	500 s

Fig. 17: Evaluation results.

Figure 17 summarizes the results of the evaluation with LoopResilience configured to use the boxes (LoopResilience-Boxes) or polyhedra (LoopResilience-Polyhedra) abstract domain to manage decision tree constraints. We configured the analysis to perform widening after two iterations, and we left the extension with lexicographic functions disabled. We used a timeout of 120s. Activating the extension or increasing the widening delay makes no difference on these benchmarks aside from an increased execution time. As expected, proving Termination Resilience instead of Termination (i.e., Termination Resilience where all variables are considered externally-controlled) considerably reduces the number of alarms, up to 62% using decision tree with polyhedral constraints. This shows that that our termination resilience analysis is an effective way to triage and prioritize alarms related to program non-termination.

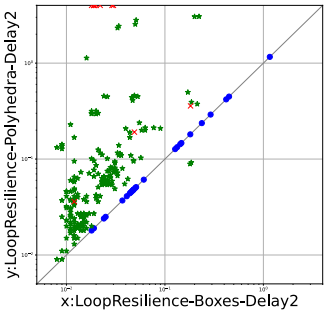


Fig. 18: Boxes vs Polyhedra.

The full comparison between LoopResilience-Boxes and LoopResilience-Polyhedra is shown in Figure 18, where green stars ($\star$) label test cases where the configurations have the same precision, while red crosses ($\times$) and blue circles ($\bullet$) label test cases where LoopResilience-Polyhedra and LoopResilience-Boxes are more precise, respectively. The presence of blue circles may be surprising: in few test cases, using a more precise numerical domain to manage decision

tree constraints actually causes a loss of precision when is widening performed (because the widening heuristic leverages adjacent decision tree partitions, and more partitions are formed with more precise decision tree constraints). We leave the implementation of better widening heuristics [8] for future work.

Comparisons with Pulse-infinite, Dynamite and Ultimate-Automizer. To evaluate the alarms raised by LoopResilience in comparison to other tools, we reused the non-termination benchmark on which Pulse was previously compared to Dynamite [20]. The programs in this benchmark are using non-linear arithmetic and are extracted from the *termination-nla* category of SV-COMP. We excluded 9 out of 40 non-terminating programs that use unsigned int, since we assume a mathematical integer semantics. As in the previous experimental evaluation, for each test filename.c, we generated the associated files for each possible combinations of variable initialization giving us 86 programs to test (with an average of 54 lines of code).

In addition to the result of [28], we ran LoopResilience and Ultimate-Automizer [7], and we report in Figure 19 whether each tool raised an alarm or not. In Figure 19, suffixes are generated by the regular expression $-[\mathrm{r}+\mathrm{i}]^*$ and describe the sequence of variable initialization. For instance the test filename-ri.c is the test filename.c where we used a non-deterministic choice ($[-\infty,+\infty]$) for the first initialization and input for the second. A true non-termination alarm (find by Pulse, Dynamite or UAutomizer) is denoted ⚠, an empty cell indicates that the tool is inconclusive. For LoopResilience ✓ means that it have proven Termination Resilience (returns true), while ❗(FP) and ❗(TP) denotes the cases where LoopResilience raised a false alarm and where it raised a true alarm, respectively. From this evaluation, we conclude that our prototype LoopResilience is able to reduce the alarms raised by state-of-the-art analyzers. More precisely, Termination Resilience is proved for 16 programs where either Pulse or Dynamite or Ultimate-Automizer found an alarm, hence LoopResilience remove 16 alarms. Moreover, Termination Resilience is proved for 4 programs where Pulse, Dynamite and Ultimate-Automizer did not find any non-termination bugs. Finally, by manual inspection, we report, we report that 57 out of the 61 alarms raised by LoopResilience are true positive. This huge number of true positives is expected. Indeed, the non-terminating programs from the SV-COMP category *termination-nla* are constructed from terminating one by making them always diverging, which is a case Robust Non-Termination where LoopResilience have to raise an alarm.

7 Related Work

Several approaches have been proposed in the literature to prove program termination [3,2,4,6,14,23,24,31,32, etc.], or the existence of diverging executions in programs [28,36,17,5,20,21,19, etc.]. However, to the best of our knowledge, none of these works studies (non-)termination in the presence of an external adversary.

Name	LoopResilience	Pulse	Dynamite	UAutomizer
bresenham1-both-nt-ii	!(TP)		⚠	
bresenham1-both-nt-ir	!(TP)		⚠	
bresenham1-both-nt-ri	!(TP)		⚠	
bresenham1-both-nt-rr	!(FP)		⚠	
cohencu1-both-nt-i	!(TP)		⚠	⚠
cohencu1-both-nt-r	!(TP)		⚠	⚠
cohencu2-both-nt-i	!(TP)			
cohencu2-both-nt-r	!(TP)			
cohencu3-both-nt-i	!(TP)			
cohencu3-both-nt-r	!(TP)			
cohencu4-both-nt-i	!(TP)			
cohencu4-both-nt-r	!(TP)			
cohencu5-both-nt-i	!(TP)		⚠	
cohencu5-both-nt-r	!(TP)		⚠	
dijkstra1-both-nt-i	!(TP)	⚠		⚠
dijkstra1-both-nt-r	✓	⚠		⚠
dijkstra2-both-nt-i	!(TP)	⚠		⚠
dijkstra2-both-nt-r	✓	⚠		⚠
dijkstra3-both-nt-i	!(TP)	⚠		⚠
dijkstra3-both-nt-r	✓	⚠		⚠
dijkstra4-both-nt-i	!(TP)	⚠		⚠
dijkstra4-both-nt-r	✓	⚠		⚠
dijkstra5-both-nt-i	!(TP)	⚠		⚠
dijkstra5-both-nt-r	✓	⚠		⚠
dijkstra4-both-nt-i	!(TP)	⚠		⚠
dijkstra6-both-nt-r	✓	⚠		⚠
egcd2-both-nt-ii	!(TP)	⚠	⚠	
egcd2-both-nt-ir	✓	⚠	⚠	
egcd2-both-nt-ri	✓	⚠	⚠	
egcd2-both-nt-rr	✓	⚠	⚠	
egcd3-both-nt-ii	!(TP)	⚠	⚠	
egcd3-both-nt-ir	✓	⚠	⚠	
egcd3-both-nt-ri	!	⚠	⚠	
egcd3-both-nt-rr	✓	⚠	⚠	
egcd-both-nt-ii	!(TP)	⚠	⚠	
egcd-both-nt-ir	✓	⚠	⚠	
egcd-both-nt-ri	✓	⚠	⚠	
egcd-both-nt-rr	✓	⚠	⚠	
freire1-both-nt-i	!(TP)	⚠		
freire1-both-nt-r	!(TP)	⚠		
geo1-both-nt-ii	!(TP)		⚠	
geo1-both-nt-ir	!(TP)		⚠	

Name	LoopResilience	Pulse	Dynamite	UAutomizer
geo1-both-nt-ri	!(TP)			⚠
geo1-both-nt-rr	!(TP)			⚠
geo2-both-nt-ii	!(TP)			⚠
geo2-both-nt-ir	!(TP)			⚠
geo2-both-nt-ri	!(TP)			⚠
geo2-both-nt-rr	!(TP)			⚠
geo3-both-nt-ii	!(TP)			⚠
geo3-both-nt-ir	!(TP)			⚠
geo3-both-nt-ri	!(TP)			⚠
geo3-both-nt-rr	!(TP)			⚠
hard2-both-nt-ii	!(TP)			⚠
hard2-both-nt-r	!(TP)			⚠
hard-both-nt-ii	!(TP)	⚠		⚠
hard-both-nt-ir	✓	⚠		⚠
hard-both-nt-ri	!(TP)	⚠		⚠
hard-both-nt-rr	✓	⚠		⚠
prod4br-both-nt-ii	!(TP)			
prod4br-both-nt-ir	✓			
prod4br-both-nt-ri	!(TP)			
prod4br-both-nt-rr	✓			
prodbin-both-nt-ii	!(TP)			
prodbin-both-nt-ir	✓			
prodbin-both-nt-ri	!(TP)			
prodbin-both-nt-rr	✓			
ps2-both-nt-i	!(TP)			⚠
ps2-both-nt-r	!(TP)			⚠
ps3-both-nt-i	!(TP)			
ps3-both-nt-r	!(TP)			
ps4-both-nt-i	!(TP)			⚠
ps4-both-nt-r	!(TP)			⚠
ps5-both-nt-i	!(TP)			
ps5-both-nt-r	!(TP)			
ps6-both-nt-i	!(TP)			
ps6-both-nt-r	!(TP)			
sqrt1-both-nt-i	!(TP)			⚠
sqrt1-both-nt-r	!(TP)			⚠
sqrt2-both-nt-ii	!(TP)			
sqrt2-both-nt-ir	!(FP)			
sqrt2-both-nt-ri	!(FP)			
sqrt2-both-nt-rr	!(FP)			

Fig. 19: Comparisons of the alarms raised by each tool

Our work is inspired by the work of Girol et al. [15,16], where they introduce the notion of Robust Reachability of a *safety* bug. They additionally propose symbolic execution and bounded model checking techniques to find robustly reachable bugs. Instead, we introduce the notion of Robust Non-Termination, the robust occurrence of a *liveness* bug, and its negation Termination Resilience. We propose a static analysis approach to verify Termination Resilience and detect potential Robust Non-Termination bugs.

In a similar spirit as Girol et al., Parolini and Miné have introduced the notion of *safety* Non-Exploitability [27,26] and proposed a static analysis, combining taint and reachable values information, to prove that an external adversary cannot trigger or silence a safety bug in a program. In our work, we focus on *liveness* bugs (notably, non-termination bugs) and we are only concerned with external adversaries triggering these bugs, not silencing them. We believe that a modification of our termination resilience semantics to reason about adversaries causing unintended program termination is possible. We leave this investigation for future work. We also observe that the analysis of Parolini and Miné only supports an attacker that has full knowledge of the non-deterministic choices made by a program, while our semantic and static analysis frameworks naturally accommodate different attacker models, including a blind attacker that makes their value choices before observing the non-deterministic choices (as in the programs in Figure 4).

Moussaoui-Remil et al., proposed an analysis operating on decision trees to infer minimum sets of variables that need to be constrained in order to ensure a CTL property [25]. We could leverage it to infer minimum sets of externally-controlled variables that we need to constrain to ensure termination resilience.

Finally, in our abstract termination resilience semantic, we implicitly use a coarse-grained notion of tainted expressions: all expressions, except non-deterministic choices, are considered as tainted. More precise abstract taint semantics have recently been proposed in the literature [27,1,22]. As explained in Section 5, leveraging them to extend the use of the resilience join in the abstract termination resilience semantics requires the development of a more precise under-approximation of the weakest liberal precondition of program statement. Moreover, contrary to ours, these analyses do not handle non-determinism natively in their definition (cf. Equation 8). Should the need arise, e.g., to analyze more complex programs than those in our benchmark, we would need to adapt these analyses to be able to soundly leverage them in our work.

8 Conclusion and Future Work

In this paper, we have proposed a novel property, Termination Resilience, and an abstract interpretation-based static analysis to infer sufficient preconditions ensuring it. To this end, we have enhanced an existing abstract domain representing piecewise-defined functions with non-trivial transformer operators. Constructing a benchmark from SV-COMP 2024 and from the state-of-the-art, we have shown that our static analysis is able to reduce by up to 62% the alarms related to program non-termination.

For future work, we plan to investigate on how we can adapt our semantics to obtain a non-exploitability analysis for liveness properties. Another interesting extension of our work will be an analysis for the resilience of CTL properties of programs [35]. Finally, we plan to extend the decision tree abstract domain to analyze data structure-manipulating programs.

References

1. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014.
2. Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: exchanging verification results between verifiers. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 326–337. ACM, 2016.

3. Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. Witness validation and stepwise testification across software verifiers. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 721–733. ACM, 2015.
4. Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety verification and refutation by k-invariants and k-induction. In Sandrine Blazy and Thomas P. Jensen, editors, *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2015.
5. Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Dorde Zikelic. Proving non-termination by program reversal. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1033–1048. ACM, 2021.
6. Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. Bit-precise procedure-modular termination analysis. *ACM Trans. Program. Lang. Syst.*, 40(1):1:1–1:38, 2018.
7. Yu-Fang Chen, Matthias Heizmann, Ondřej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. Advanced automata-based algorithms for program termination checking. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 135–150, 2018.
8. Nathanaël Courant and Caterina Urban. Precise Widening Operators for Proving Termination by Abstract Interpretation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 136–152, 2017.
9. Patrick Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.
10. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.
11. Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
12. Patrick Cousot and Radhia Cousot. An Abstract Interpretation Framework for Termination. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 245–258. ACM, 2012.
13. Robert W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
14. Mirco Giacobbe, Daniel Kroening, and Julian Parsert. Neural termination analysis. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 633–645. ACM, 2022.

15. Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 669–693. Springer, 2021.
16. Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Introducing robust reachability. *Formal Methods Syst. Des.*, 63(1):206–234, 2024.
17. Jera Hensel, Constantin Mensendiek, and Jürgen Giesl. Aprove: Non-termination witnesses for C programs - (competition contribution). In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 403–407. Springer, 2022.
18. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.
19. Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using max-smt. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 779–796. Springer, 2014.
20. Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. Dynamite: dynamic termination and non-termination proofs. *Proc. ACM Program. Lang.*, 4(OOPSLA):189:1–189:30, 2020.
21. Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 489–498. ACM, 2015.
22. Francesco Logozzo and Ibrahim Mohamed. *How to Make Taint Analysis Precise*, pages 43–55. Springer Nature Singapore, Singapore, 2023.
23. Viktor Malík, Frantisek Necas, Peter Schrammel, and Tomás Vojnar. 2ls: Arrays and loop unwinding - (competition contribution). In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 529–534. Springer, 2023.
24. Ravindra Metta, Hrishikesh Karmarkar, Kumar Madhukar, R. Venkatesh, and Supratik Chakraborty. PROTON: probes for termination or not (competition contribution). In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11,*

2024, Proceedings, Part III, volume 14572 of *Lecture Notes in Computer Science*, pages 393–398. Springer, 2024.
25. Naïm Moussaoui Remil, Caterina Urban, and Antoine Miné. Automatic detection of vulnerable variables for CTL properties of programs. In Nikolaj S. Bjørner, Marijn Heule, and Andrei Voronkov, editors, *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024*, volume 100 of *EPiC Series in Computing*, pages 116–126. EasyChair, 2024.
26. Francesco Parolini. *Static Analysis for Security Properties of Software by Abstract Interpretation. (Analyse statique des propriétés de sécurité des logiciels par interprétation abstraite).* PhD thesis, Sorbonne University, Paris, France, 2024.
27. Francesco Parolini and Antoine Miné. Sound Abstract Nonexploitability Analysis. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*, volume 14500 of *Lecture Notes in Computer Science*, pages 314–337. Springer, 2024.
28. Azalea Raad, Julien Vanegue, and Peter W. O'Hearn. Non-termination proving at scale. *Proc. ACM Program. Lang.*, 8(OOPSLA2):246–274, 2024.
29. Xiuhan Shi, Xiaofei Xie, Yi Li, Yao Zhang, Sen Chen, and Xiaohong Li. Large-scale analysis of non-termination bugs in real-world OSS projects. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 256–268. ACM, 2022.
30. Alan Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.
31. Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs (Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes).* PhD thesis, École Normale Supérieure, Paris, France, 2015.
32. Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. Synthesizing ranking functions from bits and pieces. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2016.
33. Caterina Urban and Antoine Miné. A Decision Tree Abstract Domain for Proving Conditional Termination. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014.
34. Caterina Urban and Antoine Miné. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 412–431. Springer, 2014.
35. Caterina Urban, Samuel Ueltschi, and Peter Müller. Abstract interpretation of CTL properties. In Andreas Podelski, editor, *Static Analysis - 25th International*

Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings, volume 11002 of *Lecture Notes in Computer Science*, pages 402–422. Springer, 2018.

36. Yao Zhang, Xiaofei Xie, Yi Li, Sen Chen, Cen Zhang, and Xiaohong Li. Endwatch: A practical method for detecting non-termination in real-world software. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 686–697. IEEE, 2023.

A Arithmetic Expressions Semantics

The semantics $[\![\mathsf{e}]\!] \colon \mathcal{E} \rightharpoonup \mathcal{P}(\mathbb{Z})$ of an expression $\mathsf{e} \in \text{AExp} \cup \{\mathsf{input}, [c_1, c_2]\}$ is defined as follows:

$$
\begin{aligned}
[\![x]\!]\rho &\triangleq \{\rho(x)\} \\
[\![c]\!]\rho &\triangleq \{c\} \\
[\![-\mathsf{a}]\!]\rho &\triangleq \{-n \mid n \in [\![\mathsf{a}]\!]\rho\} \\
[\![\mathsf{a}_1 \diamond \mathsf{a}_2]\!]\rho &\triangleq \{n_1 \diamond n_2 \mid n_1 \in [\![\mathsf{a}_1]\!]\rho, n_2 \in [\![\mathsf{a}_2]\!]\rho\} \\
\\
[\![\mathsf{input}]\!]\rho &\triangleq \mathbb{Z} \\
\\
[\![[c, c']]\!]\rho &\triangleq \{z \in \mathbb{Z} \mid c \leq z \leq c'\}
\end{aligned}
$$

B A More Precise Taint Semantics

In Section 4, the definition of a taint arithmetic expression (cf: Equation 8) was based on $\mathcal{E}$. This definition is too coarse as it considers any environment as possible for the arithmetic expression semantics. In order to improve it, we consider a new parameter R corresponding to a set of reachable environments before the arithmetic expressions. This let us define a more precise taint semantics of arithmetic expressions as follows:

$$
\mathfrak{T}[\![\mathsf{a}]\!](\mathrm{T}, R) \triangleq \exists \rho \in R, \exists V \in \mathbb{Z}^{\mathbb{X}} : [\![\mathsf{a}]\!]\rho = V \wedge \forall \rho' \neq_{\mathrm{T}} \rho, [\![\mathsf{a}]\!]\rho' \neq V \tag{12}
$$

An expression is tainted if there exists a value of its semantics T that is only possible in a reachable environment $\rho \in R$ with a certain value V for the set T of tainted variables, i.e., the value of the expression depends on externally-controlled program inputs. The taint semantics based on reachable states (defined bellow) is akin to the one described in Section 4 Figure 5 with an extra computation of the reachable environments after each statement (allowing us to

call $\mathfrak{T}$ with the set of reachable environments.).

$$
\begin{aligned}
\mathfrak{T}[\![^{l}\,\mathsf{skip}]\!](\mathrm{T}, R) &\triangleq (T, R)\\
\mathfrak{T}[\![^{l}\,x := \mathsf{input}]\!](\mathrm{T}, R) &\triangleq \mathrm{T} \cup \{x\}\\
\mathfrak{T}[\![^{l}\,x := \mathsf{a}]\!](\mathrm{T}, R) &\triangleq \mathsf{let}\ \mathrm{T}' = \begin{cases} \mathrm{T} \cup \{x\} & \mathfrak{T}[\![\mathsf{a}]\!](\mathrm{T}, R) \\ \mathrm{T} \setminus \{x\} & \text{otherwise} \end{cases}\\
&\quad\ \mathsf{in}\\
&\quad\ \mathsf{let}\ R' = \{\rho[x \leftarrow v] \mid \rho \in R, v \in [\![\mathsf{a}]\!]\rho\}\ \mathsf{in}\\
&\quad\ \mathsf{in}\ (\mathrm{T}', R')\\
\mathfrak{T}[\![\mathsf{if}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathsf{s}\}]\!](\mathrm{T}, R) &\triangleq \mathsf{let}\ \mathrm{T}', R' \triangleq \mathfrak{T}[\![\mathsf{s}]\!](T, \{\rho \in R \mid \exists v \in [\![\mathsf{a}]\!]\rho, v \bowtie 0\})\ \mathsf{in}\\
&\quad \begin{cases} \mathrm{T}' \cup \textsc{assigned}(R') \cup \mathrm{T}, R \cup R' & \mathfrak{T}[\![\mathsf{a}]\!](\mathrm{T}, R) \\ \mathrm{T}' \cup \mathrm{T}, R \cup R' & \text{otherwise} \end{cases}\\
\mathfrak{T}[\![\mathsf{s}_1; \mathsf{s}_2]\!](\mathrm{T}, R) &\triangleq \mathfrak{T}[\![\mathsf{s}_2]\!](\mathfrak{T}[\![\mathsf{s}_1]\!](\mathrm{T}, R))\\
\mathfrak{T}[\![\mathsf{while}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathsf{s}\}]\!](\mathrm{T}, R) &\triangleq \mathsf{lfp}^{\subseteq}_{\mathrm{T}} \mathfrak{T}[\![\mathsf{if}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathsf{s}\}]\!](X, R)
\end{aligned}
$$

[c12] **CU**, Peter Müller

An Abstract Interpretation Framework for Input Data Usage

In 27th EuropeanSymposium on Programming (ESOP2018)

https://caterinaurban.github.io/publication/esop2018/

An Abstract Interpretation Framework for Input Data Usage

Caterina Urban( ) and Peter Müller

Department of Computer Science, ETH Zurich, Zurich, Switzerland
{caterina.urban,peter.mueller}@inf.ethz.ch

Abstract. Data science software plays an increasingly important role in critical decision making in fields ranging from economy and finance to biology and medicine. As a result, errors in data science applications can have severe consequences, especially when they lead to results that look plausible, but are incorrect. A common cause of such errors is when applications erroneously ignore some of their input data, for instance due to bugs in the code that reads, filters, or clusters it.

In this paper, we propose an abstract interpretation framework to automatically detect unused input data. We derive a program semantics that precisely captures data usage by abstraction of the program's operational trace semantics and express it in a constructive fixpoint form. Based on this semantics, we systematically derive static analyses that automatically detect unused input data by fixpoint approximation.

This clear design principle provides a framework that subsumes existing analyses; we show that secure information flow analyses and a form of live variables analysis can be used for data usage, with varying degrees of precision. Additionally, we derive a static analysis to detect single unused data inputs, which is similar to dependency analyses used in the context of backward program slicing. Finally, we demonstrate the value of expressing such analyses as abstract interpretation by combining them with an existing abstraction of compound data structures such as arrays and lists to detect unused chunks of the data.

1 Introduction

In the past few years, data science has grown considerably in importance and now heavily influences many domains, ranging from economy and finance to biology and medicine. As we rely more and more on data science for making decisions, we become increasingly vulnerable to programming errors.

Programming errors can cause frustration, especially when they lead to a program failure after hours of computation. However, programming errors that do not cause failures can have more serious consequences as code that produces an erroneous but plausible result gives no indication that something went wrong. A notable example is the paper "Growth in a Time of Debt" published in 2010 by economists Reinhart and Rogoff, which was widely cited in political debates and

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 683–710, 2018.
https://doi.org/10.1007/978-3-319-89884-1_24

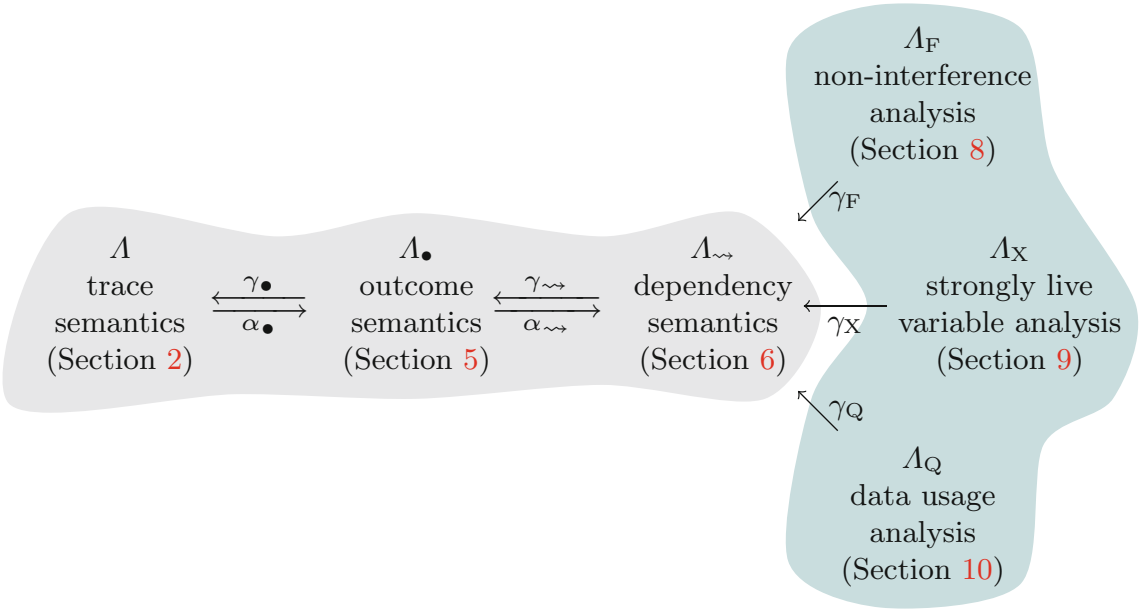


Fig. 1. Overview of the program semantics presented in the paper. The *dependency semantics*, derived by abstraction of the *trace semantics*, is sound and complete for data usage. Further sound but not complete abstractions are shown on the right.

was later demonstrated to be flawed. Notably, one of the flaws was a programming error, which *entirely excluded some data* from the analysis [23]. Its critics hold that this paper led to unjustified adoption of austerity policies for countries with various levels of public debt [30]. Programming errors in data analysis code for medical applications are even more critical [27]. It is thus paramount to achieve a high level of confidence in the correctness of data science code.

The likelihood that a programming error causes some input data to remain unused is particularly high for data science applications, where data goes through long pipelines of modules that acquire, filter, merge, and manipulate it. In this paper, we propose an abstract interpretation [14] framework to automatically detect *unused input data*. We characterize when a program uses (some of) its input data using the notion of *dependency* between the input data and the *outcome* of the program. Our notion of dependency accounts for non-determinism and non-termination. Thus, it encompasses notions of dependency that arise in many different contexts, such as secure information flow and program slicing [1], as well as provenance or lineage analysis [9], to name a few.

Following the theory of abstract interpretation [12], we systematically derive a new program semantics that precisely captures exactly the information needed to reason about input data usage, abstracting away from irrelevant details about the program behavior. Figure 1 gives an overview of our approach. The semantics is first expressed in a constructive fixpoint form over *sets of sets of traces*, by partitioning the operational trace semantics of a program based on its outcome (cf. *outcome semantics* in Fig. 1), and a further abstraction ignores intermediate state computations (cf. *dependency semantics* in Fig. 1). Starting the development of the semantics from the operational trace semantics enables a

uniform mathematical reasoning about programs semantics and program properties (Sect. 3). In particular, since input data usage is not a trace property or a subset-closed property [11] (Sect. 4), we show that a formulation of the semantics using sets of sets of traces is necessary for a sound validation of input data usage via fixpoint approximation [28].

This clear design principle provides a unifying framework for reasoning about existing analyses based on dependencies. We survey existing analyses and identify key design decisions that limit or facilitate their applicability to input data usage, and we assess their precision. We show that non-interference analyses [6] are sound for proving that a *terminating* program does not use *any* of its input data; although this is too strong a property in general. We prove that strongly live variable analysis [20] is sound for data usage even for non-terminating programs, albeit it is imprecise with respect to implicit dependencies between program variables. We then derive a more precise static analysis similar to dependency analyses used in the context of backward program slicing [37]. Finally, we demonstrate the value of expressing these analyses as abstract interpretations by combining them with an existing abstraction of compound data structures such as arrays and lists [16]. This allows us to detect unused chunks of the input data, and thus apply our work to realistic data science applications.

2 Trace Semantics

The *semantics* of a program is a mathematical characterization of its behavior when executed for all possible input data. We model the operational semantics of a program as a *transition system* $\langle \Sigma, \tau \rangle$ where Σ is a (potentially infinite) set of program states and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the possible transitions between states [12,14]. Note that this model allows representing programs with (possibly unbounded) non-determinism. The set $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \notin \tau\}$ is the set of *final states* of the program.

In the following, let $\Sigma^n \stackrel{\text{def}}{=} \{s_0 \cdots s_{n-1} \mid \forall i < n : s_i \in \Sigma\}$ be the set of all sequences of exactly n program states. We write ε to denote the empty sequence, i.e., $\Sigma^0 \stackrel{\text{def}}{=} \{\varepsilon\}$. Let $\Sigma^\star \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \Sigma^n$ be the set of all finite sequences, $\Sigma^+ \stackrel{\text{def}}{=} \Sigma^\star \setminus \Sigma^0$ be the set of all non-empty finite sequences, Σ^ω be the set of all infinite sequences, $\Sigma^{+\infty} \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$ be the set of all non-empty finite or infinite sequences and $\Sigma^{\star\infty} \stackrel{\text{def}}{=} \Sigma^\star \cup \Sigma^\omega$ be the set of all finite or infinite sequences of program states. In the following, we write $\sigma\sigma'$ for the concatenation of two sequences $\sigma, \sigma' \in \Sigma^{\star\infty}$ (with $\sigma\varepsilon = \varepsilon\sigma = \sigma$, and $\sigma\sigma' = \sigma$ when $\sigma \in \Sigma^\omega$), $T^+ \stackrel{\text{def}}{=} T \cap \Sigma^+$ and $T^\omega \stackrel{\text{def}}{=} T \cap \Sigma^\omega$ for the selection of the non-empty finite sequences and the infinite sequences of $T \in \mathcal{P}(\Sigma^{\star\infty})$, and $T \mathbin{;} T' \stackrel{\text{def}}{=} \{\sigma s \sigma' \mid s \in \Sigma \wedge \sigma s \in T \wedge s\sigma' \in T'\}$ for the merging of two sets of sequences $T \in \mathcal{P}(\Sigma^+)$ and $T' \in \mathcal{P}(\Sigma^{+\infty})$, when a finite sequence in T terminates with the initial state of a sequence in T'.

Given a transition system $\langle \Sigma, \tau \rangle$, a *trace* is a non-empty sequence of program states described by the transition relation τ, that is, $\langle s, s' \rangle \in \tau$ for each pair of

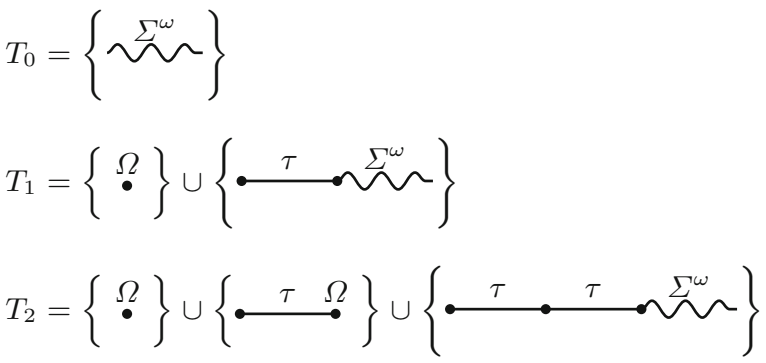


Fig. 2. First fixpoint iterates of the trace semantics Λ.

consecutive states $s, s' \in \Sigma$ in the sequence. The set of final states Ω and the transition relation τ can be understood as sets of traces of length one and length two, respectively. The *trace semantics* $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition system $\langle \Sigma, \tau \rangle$ is the union of all finite traces that are terminating with a final state in Ω, and all infinite traces. It can be expressed as a least fixpoint in the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$ [12]:

$$\begin{aligned} \Lambda &= \mathrm{lfp}^{\sqsubseteq}\, \Theta \\ \Theta(T) &\stackrel{\text{def}}{=} \Omega \cup (\tau \,;\, T) \end{aligned} \tag{1}$$

where the computational order is $T_1 \sqsubseteq T_2 \stackrel{\text{def}}{=} T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$. Figure 2 illustrates the first fixpoint iterates. The fixpoint iteration starts from the set of all infinite *sequences* of program states. At each iteration, the final program states in Ω are added to the set, and sequences already in the set are extended by prepending transitions to them. In this way, we *add* increasingly longer finite traces, and we *remove* infinite sequences of states with increasingly longer prefixes not forming traces. In particular, the i-th iterate builds all finite traces of length less than or equal to i, and selects all infinite sequences whose prefixes of length i form traces. At the limit we obtain all infinite traces and all finite traces that terminate in a final state in Ω. Note that Λ is *suffix-closed*.

The trace semantics Λ fully describes the behavior of a program. However, to reason about a particular property of a program, it is not necessary to consider all aspects of its behavior. In fact, reasoning is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. In the next sections, we define our property of interest and use abstract interpretation [14] to systematically derive, by successive abstractions of the trace semantics, a semantics that precisely captures such a property.

3 Input Data Usage

A *property* is specified by its extension, that is, the set of elements having such a property [14,15]. Thus, properties of program traces in $\Sigma^{+\infty}$ are sets of traces in

$\mathcal{P}(\Sigma^{+\infty})$, and properties of programs with trace semantics in $\mathcal{P}(\Sigma^{+\infty})$ are *sets of sets of traces* in $\mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$. Accordingly, a program P satisfies a property $\mathcal{H} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ if and only if its semantics $[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty})$ belongs to $\mathcal{H}$:

$$P \models \mathcal{H} \Leftrightarrow [\![P]\!] \in \mathcal{H} \tag{2}$$

Some program properties are defined in terms of individual program traces and can be equivalently expressed as trace properties. This is the case for the traditional safety [26] and liveness [4] properties of programs. In such a case, a program P satisfies a trace property $\mathcal{T}$ if and only if all traces in its semantics $[\![P]\!]$ belong to the property: $P \models \mathcal{T} \Leftrightarrow [\![P]\!] \subseteq \mathcal{T}$.

Program properties that establish a relation between different program traces cannot be expressed as trace properties [11]. Examples are security properties such as *non-interference* [21,35]. In this paper, we consider a closely related but more general property called *input data usage*, which expresses that *the outcome of a program does not depend on (some of) its input data*. The notion of *outcome* accounts for non-determinism as well as non-termination. Thus, our notion of dependency encompasses non-interference as well as notions of dependency that arise in many other contexts [1,9]. We further explore this in Sects. 8 to 10.

Let each program P with trace semantics $[\![P]\!]$ have a set I_P of input variables and a set O_P of output variables[1]. For simplicity, we can assume that these variables are all of the same type (e.g., boolean variables) and their values are all in a set V of possible values (e.g., $\mathrm{V} = \{\mathrm{T}, \mathrm{F}\}$ where T is the boolean value true and F is the boolean value false). Given a trace $\sigma \in [\![P]\!]$, we write $\sigma[0]$ to denote its initial state and $\sigma[\omega]$ to denote its outcome, that is, its final state if the trace is finite or $\perp$ if the trace is infinite. The input variables at the initial states of the traces of a program store the values of its input data: we write $\sigma[0](i)$ to denote the value of the input data stored in the input variable i at the initial state of the trace σ, and $\sigma_1[0] \neq_i \sigma_2[0]$ to denote that the initial states of two traces σ_1 and σ_2 disagree on the value of the input variable i but agree on the values of all other variables. The output variables at the final states of the finite traces of a program store its result: we write $\sigma[\omega](o)$ to denote the result stored in the output variable o at the final state of a finite trace σ. We can now formally define when an input variable $i \in \mathrm{I}_P$ is *unused* with respect to a program with trace semantics $[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty})$:

$$\begin{aligned} \textsc{unused}_i([\![P]\!]) \stackrel{\text{def}}{=}\ & \forall \sigma \in [\![P]\!], v \in \mathrm{V} : \sigma[0](i) \neq v \Rightarrow \\ & \exists \sigma' \in [\![P]\!] : \sigma'[0] \neq_i \sigma[0] \wedge \sigma'[0](i) = v \wedge \sigma[\omega] = \sigma'[\omega] \end{aligned} \tag{3}$$

Intuitively, an input variable i is unused if all feasible program outcomes (e.g., the outcome $\sigma[\omega]$ of a trace σ) are feasible from all possible initial values of i (i.e., for all possible initial values v of i that differ from the initial value of i in σ, there exists a trace with initial value v for i that has the same outcome $\sigma[\omega]$). In other words, the outcome of the program is the same independently of

[1] The approach can be easily extended to infinite inputs and/or outputs via abstractions such as the one later presented in Sect. 11.

```
1  english = input()
2  math = input()
3  science = input()
4  bonus = input()
5
6  passing = True
7  if not english: english = False          # english should be passing
8  if not math: passing = bonus
9  if not math: passing = bonus             # math should be science
10
11 print(passing)
```

Fig. 3. Simple program to check if a student has passed three school subjects. The programmer has made two mistakes at line 7 and at line 9, which cause the input data stored in the variables `english` and `science` to be unused.

the initial value of the input variable i. Note that this definition accounts for non-determinism (since it considers each program outcome independently) and non-termination (since a program outcome can be $\perp$).

Example 1. Let us consider the simple program P in Fig. 3. Based on the input variables `english`, `math`, and `science` (cf. lines 1–3), the program is supposed to check if a student has passed all three considered school subjects and store the result in the output variable `passing` (cf. line 11). For mathematics and science, the student is allowed a bonus based on the input variable `bonus` (cf. line 8 and 9). However, the programmer has made two mistakes at line 7 and at line 9, which cause the input variables `english` and `science` to be unused.

Let us now consider the input variable `science`. The trace semantics of the program (simplified to consider only the variables `science` and `passing`) is:

$$[\![P]\!]_{\texttt{science}} = \{(\mathrm{T}\_)\ldots(\mathrm{TT}), (\mathrm{T}\_)\ldots(\mathrm{TF}), (\mathrm{F}\_)\ldots(\mathrm{FT}), (\mathrm{F}\_)\ldots(\mathrm{FF})\}$$

where each state $(v_1 v_2)$ shows the boolean value v_1 of `science` and v_2 of `passing`, and _ denotes any boolean value. We omitted the trace suffixes for brevity. The input variable `science` is *unused*, since each result value (T or F) for `passing` is feasible from all possible initial values of `science`. Note that all other outcomes of the program (i.e., non-termination) are not feasible.

Let us now consider the input variable `math`. The trace semantics of the program (now simplified to only consider `math` and `passing`) is the following:

$$[\![P]\!]_{\texttt{math}} = \{(\mathrm{T}\_)\ldots(\mathrm{TT}), (\mathrm{F}\_)\ldots(\mathrm{FT}), (\mathrm{F}\_)\ldots(\mathrm{FF})\}$$

In this case, the input variable `math` is used since only the initial state (F_) yields the result value F for `passing` (in the final state (FF)). ■

The input data usage property $\mathcal{N}$ can now be formally defined as follows:

$$\mathcal{N} \stackrel{\text{def}}{=} \left\{ [\![P]\!] \in \mathcal{P}\left(\Sigma^{+\infty}\right) \mid \forall i \in \mathrm{I}_P : \textsc{unused}_i([\![P]\!]) \right\} \tag{4}$$

which states that the outcome of a program does not depend on *any* input data. In practice one is interested in weaker input data usage properties for a subset

J of the input variables, i.e., $\mathcal{N}_J \stackrel{\text{def}}{=} \{[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty}) \mid \forall i \in J \subseteq \mathrm{I}_P\colon \textsc{unused}_i([\![P]\!])\}$.

In the following, we use abstract interpretation to reason about input data usage. In the next section, we discuss the challenges to the application of the standard abstract interpretation framework that emerge from the fact that input data usage cannot be expressed as a trace property.

4 Sound Input Data Usage Validation

In the standard framework of abstract interpretation, one defines a semantics that precisely captures a property $\mathcal{S}$ of interest by abstraction of the trace semantics Λ [12]. Then, further abstractions $\Lambda^\sharp$ provide sound over-approximations $\gamma(\Lambda^\sharp)$ of Λ (by means of a concretization function γ): $\Lambda \subseteq \gamma(\Lambda^\sharp)$. For a *trace property*, an over-approximation $\gamma([\![P]\!]^\sharp)$ of the semantics $[\![P]\!]$ of a program P allows a sound validation of the property: since $[\![P]\!] \subseteq \gamma([\![P]\!]^\sharp)$, we have that $\gamma([\![P]\!]^\sharp) \subseteq \mathcal{S} \Rightarrow [\![P]\!] \subseteq \mathcal{S}$ and so, if $\gamma([\![P]\!]^\sharp) \subseteq \mathcal{S}$, we can conclude that $P \models \mathcal{S}$ (cf. Sect. 3). This conclusion is also valid for all other *subset-closed* properties [11]: since by definition $\gamma([\![P]\!]^\sharp) \in \mathcal{S} \Rightarrow \forall T \subseteq \gamma([\![P]\!]^\sharp)\colon T \in \mathcal{S}$, we have that $\gamma([\![P]\!]^\sharp) \in \mathcal{S} \Rightarrow [\![P]\!] \in \mathcal{S}$ (and so we can conclude that $P \models \mathcal{S}$). However, for program properties that are not subset-closed, we have that $\gamma([\![P]\!]^\sharp) \in \mathcal{S} \not\Rightarrow [\![P]\!] \in \mathcal{S}$ [28] and so we cannot conclude that $P \models \mathcal{S}$, even if $\gamma([\![P]\!]^\sharp) \in \mathcal{S}$ (cf. Eq. 2).

We have seen in the previous section that input data usage is not a trace property. The example below shows that it is *not* a subset-closed property either.

Example 2. Let us consider again the program P and its semantics $[\![P]\!]_{\texttt{science}}$ and $[\![P]\!]_{\texttt{math}}$ shown in Example 1. We have seen in Example 1 that the semantics $[\![P]\!]_{\texttt{science}}$ belongs to the data usage property $\mathcal{N}$: $[\![P]\!]_{\texttt{science}} \in \mathcal{N}$. Let us consider now the following subset T of $[\![P]\!]_{\texttt{science}}$:

$$T = \{(\mathrm{T}\_)\ldots(\mathrm{TT}), (\mathrm{F}\_)\ldots(\mathrm{FT}), (\mathrm{F}\_)\ldots(\mathrm{FF})\}$$

In this case, the input variable `science` is used. Indeed, we can observe that T coincides with $[\![P]\!]_{\texttt{math}}$ (except for the considered input variable). Thus $T \notin \mathcal{N}$ even though $T \subseteq [\![P]\!]_{\texttt{science}}$. ■

Since input data usage is not subset-closed, we are in the unfortunate situation that we cannot use the standard abstract interpretation framework to soundly prove that a program does not use (some of) its input data using an over-approximation of the semantics of the program: $\gamma([\![P]\!]^\sharp) \in \mathcal{N}_J \not\Rightarrow [\![P]\!] \in \mathcal{N}_J$.

We solve this problem in the next section, by lifting the trace semantics $[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty})$ of a program P (i.e., a set of traces) to a set of sets of traces $(\!|P|\!) \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ [28]. In this setting, a program P satisfies a property $\mathcal{H}$ if and only if its semantics $(\!|P|\!)$ is a subset of $\mathcal{H}$:

$$P \models \mathcal{H} \Leftrightarrow (\!|P|\!) \subseteq \mathcal{H} \tag{5}$$

As we will explain in the next section, now an over-approximation $\gamma(\llparenthesis P \rrparenthesis^\natural)$ of $\llparenthesis P \rrparenthesis$ allows again a sound validation of the property: since $\llparenthesis P \rrparenthesis \subseteq \gamma(\llparenthesis P \rrparenthesis^\natural)$, we have that $\gamma(\llparenthesis P \rrparenthesis^\natural) \subseteq \mathcal{H} \Rightarrow \llparenthesis P \rrparenthesis \subseteq \mathcal{H}$ (and so we can conclude that $P \models \mathcal{H}$).

More specifically, in the next section, we define a program semantics $\llparenthesis P \rrparenthesis$ that precisely captures which subset J of the input variables is unused by a program P. In later sections, we present further abstractions $\llparenthesis P \rrparenthesis^\natural$ that over-approximate the subset of the input variables that *may be used* by P, and thus allows a sound validation of an *under-approximation* $J^\natural$ of J: $\gamma(\llparenthesis P \rrparenthesis^\natural) \subseteq \mathcal{N}_{J^\natural} \Rightarrow \llparenthesis P \rrparenthesis \subseteq \mathcal{N}_{J^\natural}$. In other words, this means that every input variable reported as unused by an abstraction is indeed not used by the program.

5 Outcome Semantics

We lift the trace semantics Λ to a set of sets of traces by *partitioning*. The *partitioning abstraction* $\alpha_Q \colon \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})) \to \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ of a set of traces T is:

$$\alpha_Q(T) \stackrel{\text{def}}{=} \{T \cap C \mid C \in Q\} \tag{6}$$

where $Q \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ is a *partition* of sequences of program states.

More specifically, to reason about input data usage of a program P, we lift the trace semantics $[\![P]\!]$ to $\llparenthesis P \rrparenthesis$ by partitioning it into sets of traces that yield the same program outcome. The key insight behind this idea is that, given an input variable i, the initial states of all traces in a partition give all initial values for i that yield a program outcome; the variable i is unused if and only if these initial values are all the possible values for i (or the set of values is empty because the outcome is unfeasible, cf. Eq. 3). Thus, if the trace semantics $[\![P]\!]$ of a program P belongs to the input data usage property $\mathcal{N}_J$, then each partition in $\llparenthesis P \rrparenthesis$ must also belong to $\mathcal{N}_J$, and vice versa: we have that $[\![P]\!] \in \mathcal{N}_J \Leftrightarrow \llparenthesis P \rrparenthesis \subseteq \mathcal{N}_J$, which is precisely what we want (cf. Eq. 5).

Let $T^+_{o=v}$ denote the subset of the finite sequences of program states in $T \in \mathcal{P}(\Sigma^{+\infty})$ with value v for the output variable o in their outcome (i.e., their final state): $T^+_{o=v} \stackrel{\text{def}}{=} \{\sigma \in T^+ \mid \sigma[\omega](o) = v\}$. We define the *outcome partition* $O \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ of sequences of program states:

$$O \stackrel{\text{def}}{=} \left\{ \Sigma^+_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V} \right\} \cup \{\Sigma^\omega\}$$

where V is the set of possible values of the output variables $o_1, \ldots, o_k$ (cf. Sect. 3). The partition contains all sets of finite sequences that agree on the values of the output variables in their outcome, and all infinite sequences of program states (i.e., all sequences with outcome $\perp$). We instantiate α_Q above with the outcome partition to obtain the *outcome abstraction* $\alpha_\bullet \colon \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})) \to \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$:

$$\alpha_\bullet(T) \stackrel{\text{def}}{=} \left\{ T^+_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V} \right\} \cup \{T^\omega\} \tag{7}$$

Example 3. The program P of Example 1 has only one output variable `passing` with boolean value T or F. Let us consider again the trace semantics $[\![P]\!]_{\texttt{math}}$ shown in Example 1. Its outcome abstraction $\alpha_\bullet([\![P]\!]_{\texttt{math}})$ is:

$$\alpha_\bullet([\![P]\!]_{\texttt{math}}) = \{\emptyset, \{(\texttt{F}\_)\dots(\texttt{FF})\}, \{(\texttt{T}\_)\dots(\texttt{TT}), (\texttt{F}\_)\dots(\texttt{FT})\}\}$$

Note that all traces with different result values for the output variable `passing` belong to different sets of traces (i.e., partitions) in $\alpha_\bullet([\![P]\!]_{\texttt{math}})$. The empty set corresponds to the (unfeasible) non-terminating outcome of the program. ■

We can now use the outcome abstraction $\alpha_\bullet$ to define the *outcome semantics* $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ as an abstraction of the trace semantics Λ:

Definition 1. *The* outcome semantics $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ *is defined as:*

$$\Lambda_\bullet \stackrel{\text{def}}{=} \alpha_\bullet(\Lambda) \tag{8}$$

where $\alpha_\bullet$ is the outcome abstraction (cf. Eq. 7) and $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ is the trace semantics (cf. Eq. 1).

The outcome semantics contains the set of all infinite traces and all sets of finite traces that agree on the value of the output variables in their outcome.

In the following, we express the outcome semantics $\Lambda_\bullet$ in a constructive fixpoint form. This allows us to later derive further abstractions of $\Lambda_\bullet$ by *fixpoint transfer* and *fixpoint approximation* [12]. Given a set of sets of traces S, we write $S^+_{o=v} \stackrel{\text{def}}{=} \{T \in S \mid T = T^+_{o=v}\}$ for the selection of the sets of traces in S that agree on the value v of the output variable o in their outcome, and $S^\omega \stackrel{\text{def}}{=} \{T \in S \mid T = T^\omega\}$ for the selection of the sets of infinite traces in S. When $S^+_{o=v}$ (resp. S^ω) contains a single set of traces T, we abuse notation and write $S^+_{o=v}$ (resp. S^ω) to also denote T. The following result gives a fixpoint definition of $\Lambda_\bullet$ in the complete lattice $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \sqsubseteq^\bullet, \sqcup^\bullet, \sqcap^\bullet, \{\Sigma^\omega, \emptyset\}, \{\emptyset, \Sigma^+\}\rangle$, where the computational order $\sqsubseteq^\bullet$ is defined (similarly to $\sqsubseteq$, cf. Sect. 2) as:

$$S_1 \sqsubseteq^\bullet S_2 \stackrel{\text{def}}{=} \bigwedge_{v_1,\dots,v_k \in \mathrm{V}} {S_1}^+_{o_1=v_1,\dots,o_k=v_k} \subseteq {S_2}^+_{o_1=v_1,\dots,o_k=v_k} \wedge S_1^\omega \supseteq S_2^\omega$$

Theorem 1. *The outcome semantics $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ can be expressed as a least fixpoint in $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \sqsubseteq^\bullet, \sqcup^\bullet, \sqcap^\bullet, \{\Sigma^\omega, \emptyset\}, \{\emptyset, \Sigma^+\}\rangle$ as:*

$$\begin{aligned} \Lambda_\bullet &= \mathrm{lfp}^{\sqsubseteq^\bullet}\, \Theta_\bullet \\ \Theta_\bullet(S) &\stackrel{\text{def}}{=} \{\Omega_{o_1=v_1,\dots,o_k=v_k} \mid v_1,\dots,v_k \in \mathrm{V}\} \uplus \{\tau\,;T \mid T \in S\} \end{aligned} \tag{9}$$

where $S_1 \uplus S_2 \stackrel{\text{def}}{=} \left\{ {S_1}^+_{o_1=v_1,\dots,o_k=v_k} \cup {S_2}^+_{o_1=v_1,\dots,o_k=v_k} \mid v_1,\dots,v_k \in \mathrm{V} \right\} \cup S_1^\omega \cup S_2^\omega$.

Figure 4 illustrates the first fixpoint iterates of the outcome semantics for a single output variable o. The fixpoint iteration starts from the partition containing the set of all infinite sequences of program states and the empty set (which

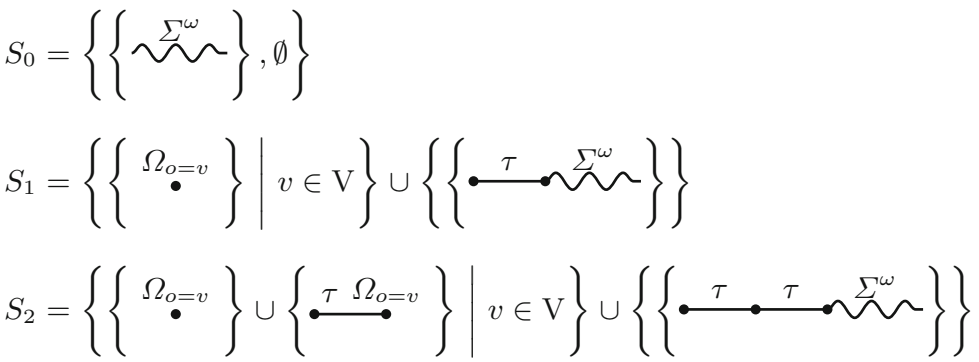


Fig. 4. First iterates of the outcome semantics $\Lambda_\bullet$ for a single output variable o.

represents an empty set of finite traces). At the first iteration, the empty set is replaced with a partition of the final states Ω based on the value v of the output variable o, while the infinite sequences are extended by prepending transitions to them (similarly to the trace semantics, cf. Eq. 1). At the next iterations, all sequences contained in each partition are further extended, and the final states that agree on the value v of o are again added to the matching set of traces that agree on v in their outcome. At the limit, we obtain a partition containing the set of all infinite traces and all sets of finite traces that agree on the value v of the output variable o in their outcome.

To prove Theorem 1 we first need to show that the outcome abstraction $\alpha_\bullet$ preserves least upper bounds of non-empty sets of sets of traces.

Lemma 1. *The outcome abstraction $\alpha_\bullet$ is Scott-continuous.*

Proof. We need to show that for any non-empty ascending chain C of sets of traces with least upper bound $\sqcup C$, we have that $\alpha_\bullet(\sqcup C) = \dot{\sqcup}\{\alpha_\bullet(T) \mid T \in C\}$, that is, $\alpha_\bullet(\sqcup C)$ is the least upper bound of $\alpha_\bullet(C)$, the image of C via $\alpha_\bullet$.

First, we know that $\alpha_\bullet$ is monotonic, i.e., for any two sets of traces T_1 and T_2 we have $T_1 \sqsubseteq T_2 \Rightarrow \alpha_\bullet(T_1) \dot{\sqsubseteq} \alpha_\bullet(T_2)$. Since $\sqcup C$ is the least upper bound of C, for any set T in C we have that $T \sqsubseteq \sqcup C$ and, since $\alpha_\bullet$ is monotonic, we have that $\alpha_\bullet(T) \dot{\sqsubseteq} \alpha_\bullet(\sqcup C)$. Thus $\alpha(\sqcup C)$ is an upper bound of $\{\alpha_\bullet(T) \mid T \in C\}$.

To show that $\alpha(\sqcup C)$ is the least upper bound of $\alpha_\bullet(C)$, we need to show that for any other upper bound U of $\alpha_\bullet(C)$ we have $\alpha_\bullet(\sqcup C) \dot{\sqsubseteq} U$. Let us assume by absurd that $\alpha_\bullet(\sqcup C) \not\dot{\sqsubseteq} U$. Then, there exists $T_1 \in \alpha_\bullet(\sqcup C)$ and $T_2 \in U$ such that $T_1 \not\sqsubseteq T_2$: $T_1^+ \supset T_2^+$ or $T_1^\omega \subset T_2^\omega$. Let us assume that $T_1^+ \supset T_2^+$. By definition of $\alpha_\bullet$, we observe that T_1 is a partition of $\dot{\sqcup} C$ and, since $\dot{\sqcup} C$ is the least upper bound of C, U cannot be an upper bound of $\alpha_\bullet(C)$ (since T_2 does not contain enough finite traces). Similarly, if $T_1^\omega \subset T_2^\omega$, then U cannot be an upper bound of $\alpha_\bullet(C)$ (since T_2 contains too many infinite traces). Thus, we must have $\alpha_\bullet(\sqcup C) \dot{\sqsubseteq} U$ and we can conclude that $\alpha(\sqcup C)$ is the least upper bound of $\alpha_\bullet(C)$. □

We can now prove Theorem 1 by Kleenian fixpoint transfer [12].

Proof (Sketch). The proof follows by Kleenian fixpoint transfer. We have that $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \sqsubseteq^{\cdot}, \dot{\cup}, \dot{\cap}, \{\Sigma^\omega, \emptyset\}, \{\emptyset, \Sigma^+\}\rangle$ is a complete lattice and that $\phi^{+\infty}$ (cf. Eq. 1) and $\Theta_\bullet$ (cf. Eq. 8) are monotonic function. Additionally, we have that the outcome abstraction $\alpha_\bullet$ (cf. Eq. 7) is Scott-continuous (cf. Lemma 1) and such that $\alpha_\bullet(\Sigma^\omega) = \{\Sigma^\omega, \emptyset\}$ and $\alpha_\bullet \circ \phi^{+\infty} = \Theta_\bullet \circ \alpha_\bullet$. Then, by Kleenian fixpoint transfer, we have that $\alpha_\bullet(\Lambda) = \alpha_\bullet(\mathrm{lfp}^{\sqsubseteq} \phi^{+\infty}) = \mathrm{lfp}^{\sqsubseteq^{\cdot}} \Theta_\bullet$. Thus, we can conclude that $\Lambda_\bullet = \mathrm{lfp}^{\sqsubseteq^{\cdot}} \Theta_\bullet$. □

Finally, we show that the outcome semantics $\Lambda_\bullet$ is sound and complete for proving that a program does not use (a subset of) its input variables.

Theorem 2. *A program does not use a subset J of its input variables if and only if its outcome semantics $\Lambda_\bullet$ is a subset of $\mathcal{N}_J$:*

$$P \models \mathcal{N}_J \Leftrightarrow \Lambda_\bullet \subseteq \mathcal{N}_J$$

Proof (Sketch). The proof follows immediately from the definition of $\mathcal{N}_J$ (cf. Eq. 3 and Sect. 4) and the definition of $\Lambda_\bullet$ (cf. Eq. 8). □

Example 4. Let us consider again the program P and its semantics $[\![P]\!]_{\texttt{science}}$ shown in Example 1. The corresponding outcome semantics $\alpha_\bullet([\![P]\!]_{\texttt{science}})$ is:

$$\alpha_\bullet([\![P]\!]_{\texttt{science}}) = \{\emptyset, \{(\mathrm{T}\_)\ldots(\mathrm{TF}), (\mathrm{F}\_)\ldots(\mathrm{FF})\}, \{(\mathrm{T}\_)\ldots(\mathrm{TT}), (\mathrm{F}\_)\ldots(\mathrm{FT})\}\}$$

Note that all sets of traces in $\alpha_\bullet([\![P]\!]_{\texttt{science}})$ belong to $\mathcal{N}_{\{\texttt{science}\}}$: the initial states of all traces in a non-empty partition contain all possible initial values (T or F) for the input variable `science`. Thus, P satisfies $\mathcal{N}_{\{\texttt{science}\}}$ and, indeed, the input variable `science` is unused by P. ■

As discussed in Sect. 4, we now can again use the standard framework of abstract interpretation to soundly over-approximate $\Lambda_\bullet$ and prove that a program does not use (some of) its input data. In the next section, we propose an abstraction that remains sound and complete for input data usage. Further sound but not complete abstractions are presented in later sections.

6 Dependency Semantics

We observe that, to reason about input data usage, it is not necessary to consider all intermediate state computations between the initial state of a trace and its outcome. Thus, we can further abstract the outcome semantics $\Lambda_\bullet$ into a set $\Lambda_{\rightsquigarrow}$ of (dependency) relations between initial states and outcomes of a set of traces.

We lift the abstraction defined for this purpose on sets of traces [12] to $\alpha_{\rightsquigarrow} : \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})) \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\perp))$ on sets of sets of traces:

$$\alpha_{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \{\{\langle \sigma[0], \sigma[\omega]\rangle \in \Sigma \times \Sigma_\perp \mid \sigma \in T\} \mid T \in S\} \tag{10}$$

where $\Sigma_\perp \stackrel{\text{def}}{=} \Sigma \cup \{\perp\}$. The *dependency abstraction* $\alpha_{\rightsquigarrow}$ ignores all intermediate states between the initial state $\sigma[0]$ and the outcome $\sigma[\omega]$ of all traces σ in

all partitions T of S. Observe that a trace σ that consists of a single state s is abstracted as a pair $\langle s, s\rangle$. The corresponding dependency concretization function $\gamma_{\rightsquigarrow} : \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\bot)) \to \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ over-approximates the original sets of traces by inserting arbitrary intermediate states:

$$\gamma_{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \left\{T \in \mathcal{P}\left(\Sigma^{+\infty}\right) \mid \{\langle \sigma[0], \sigma[\omega]\rangle \in \Sigma \times \Sigma_\bot \mid \sigma \in T\} \in S\right\} \tag{11}$$

Example 5. Let us consider again the program of Example 1 and its outcome semantics $\alpha_\bullet(\llbracket P \rrbracket_{\texttt{math}})$ shown in Example 3. Its dependency abstraction is:

$$\alpha_{\rightsquigarrow}(\alpha_\bullet(\llbracket P \rrbracket_{\texttt{math}})) = \{\emptyset, \{\langle \texttt{F}\_, \texttt{FF}\rangle\}, \{\langle \texttt{T}\_, \texttt{TT}\rangle, \langle \texttt{F}\_, \texttt{FT}\rangle\}\}$$

which explicitly ignores intermediate program states. ■

Using $\alpha_{\rightsquigarrow}$, we now define the *dependency semantics* $\Lambda_{\rightsquigarrow} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ as an abstraction of the outcome semantics $\Lambda_\bullet$.

Definition 2. *The* dependency semantics $\Lambda_{\rightsquigarrow} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ *is defined as:*

$$\Lambda_{\rightsquigarrow} \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(\Lambda_\bullet) \tag{12}$$

where $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ *is the outcome semantics (cf. Eq. 8) and* $\alpha_{\rightsquigarrow}$ *is the dependency abstraction (cf. Eq. 10).*

Neither the Kleenian fixpoint transfer nor the Tarskian fixpoint transfer can be used to obtain a fixpoint definition for the dependency semantics, but we have to proceed by union of disjoint fixpoints [12]. To this end, we observe that the outcome semantics $\Lambda_\bullet$ can be equivalently expressed as follows:

$$\begin{aligned} \Lambda_\bullet &= \Lambda_\bullet^+ \cup \Lambda_\bullet^\omega = \mathrm{lfp}^{\sqsubseteq\cdot}_{\emptyset}\ \Theta_\bullet^+ \cup \mathrm{lfp}^{\sqsubseteq\cdot}_{\{\Sigma^\omega\}}\ \Theta_\bullet^\omega \\ \Theta_\bullet^+(S) &\stackrel{\text{def}}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V}\} \uplus \{\tau\,;T \mid T \in S\} \\ \Theta_\bullet^\omega(S) &\stackrel{\text{def}}{=} \{\tau\,;T \mid T \in S\} \end{aligned} \tag{13}$$

where $\Lambda_\bullet^+$ and $\Lambda_\bullet^\omega$ separately compute the set of all sets of finite traces that agree on their outcome, and the set of all infinite traces, respectively.

In the following, given a set of traces $T \in \mathcal{P}(\Sigma^{+\infty})$ and its dependency abstraction $\alpha_{\rightsquigarrow}(T)$, we abuse notation and write T^+ (resp. T^ω) to also denote $\alpha_{\rightsquigarrow}(T)^+ \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(T) \cap (\Sigma \times \Sigma)$ (resp. $\alpha_{\rightsquigarrow}(T)^\omega \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(T) \cap (\Sigma \times \{\bot\})$). Similarly, we reuse the symbols for the computational order $\sqsubseteq\cdot$, least upper bound $\sqcup\cdot$, and greatest lower bound $\sqcap\cdot$, instead of their abstractions. We can now use the Kleenian and Tarskian fixpoint transfer to separately derive fixpoint definitions of $\alpha_{\rightsquigarrow}(\Lambda_\bullet^+)$ and $\alpha_{\rightsquigarrow}(\Lambda_\bullet^\omega)$ in $\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\bot)), \sqsubseteq\cdot, \sqcup\cdot, \sqcap\cdot, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$.

Lemma 2. *The abstraction* $\Lambda_{\rightsquigarrow}^+ \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(\Lambda_\bullet^+) \in \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$ *can be expressed as a least fixpoint in* $\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\bot)), \sqsubseteq\cdot, \sqcup\cdot, \sqcap\cdot, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$ *as:*

$$\begin{aligned} \Lambda_{\rightsquigarrow}^+ &= \mathrm{lfp}^{\sqsubseteq\cdot}_{\{\emptyset\}}\ \Theta_{\rightsquigarrow}^+ \\ \Theta_{\rightsquigarrow}^+(S) &\stackrel{\text{def}}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \times \Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V}\} \uplus \{\tau \circ R \mid R \in S\} \end{aligned} \tag{14}$$

Proof (Sketch). By Kleenian fixpoint transfer (cf. Theorem 17 in [12]). □

Lemma 3. *The abstraction* $\Lambda^{\omega}_{\rightsquigarrow} \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(\Lambda^{\omega}_{\bullet}) \in \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$ *can be expressed as a least fixpoint in* $\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_{\bot})), \dot{\sqsubseteq}, \dot{\cup}, \dot{\cap}, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$ *as:*

$$\begin{aligned} \Lambda^{\omega}_{\rightsquigarrow} &= \mathrm{lfp}^{\dot{\sqsubseteq}}_{\{\Sigma\times\{\bot\}\}} \Theta^{\omega}_{\rightsquigarrow} \\ \Theta^{\omega}_{\rightsquigarrow}(S) &\stackrel{\text{def}}{=} \{\tau \circ R \mid R \in S\} \end{aligned} \tag{15}$$

Proof (Sketch). By Tarskian fixpoint transfer (cf. Theorem 18 in [12]). □

The fixpoint iteration for $\Lambda^{+}_{\rightsquigarrow}$ starts from the set containing only the empty relation. At the first iteration, the empty relation is replaced by all relations between pairs of final states that agree on the values of the output variables. At each next iteration, all relations are combined with the transition relation to obtain relations between initial and final states of increasingly longer traces. At the limit, we obtain the set of all relations between the initial and the final states of a program that agree on the final value of the output variables. The fixpoint iteration for $\Lambda^{\omega}_{\rightsquigarrow}$ starts from the set containing (the set of) all pairs of states and the $\bot$ outcome, and each iteration discards more and more pairs with initial states that do not belong to infinite traces of the program.

Now we can use Lemmas 2 and 3 to express the dependency semantics $\Lambda_{\rightsquigarrow}$ in a constructive fixpoint form (as the union of $\Lambda^{+}_{\rightsquigarrow}$ and $\Lambda^{\omega}_{\rightsquigarrow}$).

Theorem 3. *The dependency semantics* $\Lambda_{\rightsquigarrow} \in \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_{\bot}))$ *can be expressed as a least fixpoint in* $\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_{\bot})), \dot{\sqsubseteq}, \dot{\cup}, \dot{\cap}, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$ *as:*

$$\begin{aligned} \Lambda_{\rightsquigarrow} &= \Lambda^{+}_{\rightsquigarrow} \cup \Lambda^{\omega}_{\rightsquigarrow} = \mathrm{lfp}^{\dot{\sqsubseteq}}_{\{\Sigma\times\{\bot\},\emptyset\}} \Theta_{\rightsquigarrow} \\ \Theta_{\rightsquigarrow}(S) &\stackrel{\text{def}}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \times \Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V}\} \uplus \{\tau \circ R \mid R \in S\} \end{aligned} \tag{16}$$

Proof (Sketch). The proof follows immediately from Lemmas 2 and 3. □

Finally, we show that the dependency semantics $\Lambda_{\rightsquigarrow}$ is sound and complete for proving that a program does not use (a subset of) its input variables.

Theorem 4. *A program does not use a subset* J *of its input variables if and only if the image via* $\gamma_{\rightsquigarrow}$ *of its dependency semantics* $\Lambda_{\rightsquigarrow}$ *is a subset of* $\mathcal{N}_J$*:*

$$P \models \mathcal{N}_J \Leftrightarrow \gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}_J$$

Proof (Sketch). The proof follows from the definition of $\Lambda_{\rightsquigarrow}$ (cf. Eq. 12) and $\gamma_{\rightsquigarrow}$ (cf. Eq. 11), and from Theorem 2. □

Example 6. Let us consider again the program P and its outcome semantics $\alpha_{\bullet}([\![P]\!]_{\texttt{science}})$ from Example 4. The corresponding dependency semantics is:

$$\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})) = \{\emptyset, \{\langle \mathrm{T}\_, \mathrm{TF}\rangle, \langle \mathrm{F}\_, \mathrm{FF}\rangle\}, \{\langle \mathrm{T}\_, \mathrm{TT}\rangle, \langle \mathrm{F}\_, \mathrm{FT}\rangle\}\}$$

and, by definition of $\gamma_{\rightsquigarrow}$, we have that its concretization $\gamma_{\rightsquigarrow}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})))$ is an over-approximation of $\alpha_{\bullet}([\![P]\!]_{\texttt{science}})$. In particular, since intermediate state computations are irrelevant for deciding the input data usage property, all sets of traces in $\gamma_{\rightsquigarrow}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})))$ are over-approximations of exactly one set in $\alpha_{\bullet}([\![P]\!]_{\texttt{science}})$ with the same set of initial states and outcome. Thus, in this case, we can observe that all sets of traces in $\gamma_{\rightsquigarrow}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})))$ belong to $\mathcal{N}_{\{\texttt{science}\}}$ and correctly conclude that P does not use the variable `science`. ■

At this point we have a sound and complete program semantics that captures only the minimal information needed to decide which input variables are unused by a program. In the rest of the paper, we present various static analyses for input data usage by means of sound abstractions of this semantics, which *under-approximate* (resp. over-approximate) the subset of the input variables that are *unused* (resp. used) by a program.

7 Input Data Usage Abstractions

We introduce a simple sequential programming language with boolean variables, which we use for illustration throughout the rest of the paper:

$$e ::= v \mid x \mid \texttt{not}\ e \mid e\ \texttt{and}\ e \mid e\ \texttt{or}\ e \qquad \text{(expressions)}$$
$$s ::= \texttt{skip} \mid x = e \mid \texttt{if}\ e\texttt{:}\ s\ \texttt{else:}\ s \mid \texttt{while}\ e\texttt{:}\ s \mid s\ s \qquad \text{(statements)}$$

where v ranges over boolean values, and x ranges over program variables. The `skip` statement, which does nothing, is a placeholder useful, for instance, for writing a conditional `if` statement without an `else` branch: `if` e`:` s `else: skip`. In the following, we often simply write `if` e`:` s instead of `if` e`:` s `else: skip`. Note that our work is not limited by the choice of a particular programming language, as the formal treatment in previous sections is language independent.

In Sects. 8 and 9, we show that existing static analyses based on dependencies [6,20] are abstractions of the dependency semantics $\Lambda_{\rightsquigarrow}$. We define each abstraction $\Lambda^{\natural}$ over a partially ordered set $\langle \mathcal{A}, \sqsubseteq_A \rangle$ called *abstract domain*. More specifically, for each program statement s, we define a *transfer function* $\Theta^{\natural}[\![s]\!] : \mathcal{A} \to \mathcal{A}$, and the abstraction $\Lambda^{\natural}$ is the composition of the transfer functions of all statements in a program. We derive a more precise static analysis similar to dependency analyses used for program slicing [37] in Sect. 10. Finally, Sect. 11 demonstrates the value of expressing such analyses as abstract domains by combining them with an existing abstraction of compound data structures such as arrays and lists [16] to detect unused chunks of input data.

8 Secure Information Flow Abstractions

Secure information flow analysis [18] aims at proving that a program will not leak sensitive information. Most analyses focus on proving *non-interference* [35] by classifying program variables into different security levels [17], and ensuring the

absence of information flow from variables with higher security level to variables with lower security level. The most basic classification comprises a low security level L, and a high security level H: program variables classified as L are public information, while variables classified as H are private information.

In our context, if we classify input variables as H and all other variables as L, possiblistic non-interference [21] coincides with the input data usage property $\mathcal{N}$ (cf. Eq. 4) *restricted to consider only terminating programs*. However, in general, (possibilistic) non-interference is too strong for our purposes as it requires that *none* of the input variables is used by a program. We illustrate this using as an example a non-interference analysis recently proposed by Assaf et al. [6] that is conveniently formalized in the framework of abstract interpretation. We briefly present here a version of the originally proposed analysis, simplified to consider only the security levels L and H, and we point out the significance of the definitions for input data usage.

Let $\mathcal{L} \stackrel{\text{def}}{=} \{L, H\}$ be the set of security levels, and let the set X of all program variables be partitioned into a set X_L of variables classified as L and a set X_H of variables classified as H (i.e., the input variables). A dependency constraint $L \rightsquigarrow x$ expresses that the current value of the variable x depends only on the initial values of variables having at most security level L (i.e., it does not depend on the initial value of any of the input variables). The non-interference analysis Λ_{F} proposed by Assaf et al. is a *forward analysis* in the lattice $\langle \mathcal{P}(\mathrm{F}), \sqsubseteq_{\mathrm{F}}, \sqcup_{\mathrm{F}} \rangle$ where $\mathrm{F} \stackrel{\text{def}}{=} \{L \rightsquigarrow x \mid x \in \mathrm{X}\}$ is the set of all dependency constraints, $S_1 \sqsubseteq_{\mathrm{F}} S_2 \stackrel{\text{def}}{=} S_1 \supseteq S_2$, and $S_1 \sqcup_{\mathrm{F}} S_2 \stackrel{\text{def}}{=} S_1 \cap S_2$. The transfer function $\Theta_{\mathrm{F}}[\![s]\!] : \mathcal{P}(\mathrm{F}) \to \mathcal{P}(\mathrm{F})$ for each statement s in our simple programming language is defined as follows:

$$
\begin{aligned}
\Theta_{\mathrm{F}}[\![\texttt{skip}]\!](S) &\stackrel{\text{def}}{=} S \\
\Theta_{\mathrm{F}}[\![x = e]\!](S) &\stackrel{\text{def}}{=} \{L \rightsquigarrow y \in S \mid y \neq x\} \cup \{L \rightsquigarrow x \mid \mathcal{V}_{\mathrm{F}}[\![e]\!]S\} \\
\Theta_{\mathrm{F}}[\![\texttt{if } e\texttt{: } s_1 \texttt{ else: } s_2]\!](S) &\stackrel{\text{def}}{=} \begin{cases} \Theta_{\mathrm{F}}[\![s_1]\!](S) \sqcup_{\mathrm{F}} \Theta_{\mathrm{F}}[\![s_2]\!](S) & \text{if } \mathcal{V}_{\mathrm{F}}[\![e]\!]S \\ \{L \rightsquigarrow x \in S \mid x \notin \mathrm{W}(s_1) \cup \mathrm{W}(s_2)\} & \text{otherwise} \end{cases} \\
\Theta_{\mathrm{F}}[\![\texttt{while } e\texttt{: } s]\!](S) &\stackrel{\text{def}}{=} \mathrm{lfp}^{\sqsubseteq_{\mathrm{F}}}_{S}\ \Theta_{\mathrm{F}}[\![\texttt{if } e\texttt{: } s \texttt{ else: skip}]\!] \\
\Theta_{\mathrm{F}}[\![s_1\ s_2]\!](S) &\stackrel{\text{def}}{=} \Theta_{\mathrm{F}}[\![s_2]\!] \circ \Theta_{\mathrm{F}}[\![s_1]\!](S)
\end{aligned}
$$

where $\mathrm{W}(s)$ denotes the set of variables modified by the statement s, and $\mathcal{V}_{\mathrm{F}}[\![e]\!]S$ determines whether a set of dependencies S guarantees that the expression e has a unique value independently of the initial value of the input variables. For a variable x, $\mathcal{V}_{\mathrm{F}}[\![x]\!]S$ is true if and only if $L \rightsquigarrow x \in S$. Otherwise, $\mathcal{V}_{\mathrm{F}}[\![e]\!]S$ is defined recursively on the structure of e, and it is always true for a boolean value v [6]. An assignment $x = e$ discards all dependency constraints related to the assigned variable x, and adds constraints $L \rightsquigarrow x$ if e has a unique value independently of the initial values of the input variables. This captures an *explicit flow* of information between e and x. A conditional statement $\texttt{if } e\texttt{: } s_1 \texttt{ else: } s_2$ joins the dependency constraints obtained from s_1 and s_2, if e does not depend

on the initial values of the input variables (i.e., $\mathcal{V}_{\mathrm{F}}[\![e]\!]S$ is true). Otherwise, it discards all dependency constraints related to the variables modified in either of its branches. This captures an *implicit flow* of information from e. The initial set of dependencies contains a constraint $L \rightsquigarrow x$ for each variable x that is not an input variable. We exemplify the analysis below.

Example 7. Let us consider again the program P from Example 1 (stripped of the `input` and `print` statements, which are not present in our simple language):

```
1 passing = True
2 if not english: english = False              # english should be passing
3 if not math: passing = bonus
4 if not math: passing = bonus                 # math should be science
```

The analysis begins from the set of dependency constraints $\{L \rightsquigarrow \texttt{passing}\}$, which classifies input variables as H and all other variables as L. The assignment at line 1 leaves the set unchanged as the value of the expression `True` on the right-hand side of the assignment does not depend on the initial value of the input variables. The set remains unchanged by the conditional statement at line 2, even though the boolean condition depends on the input variable `english`, because the variable `passing` is not modified. Finally, at line 3 and 4, the analysis captures an explicit flow of information from the input variable `bonus` and an implicit flow of information from the input variable `math`. Thus, the set of dependency constraints becomes empty at line 3, and remains empty at line 4.

Observe that, in this case, non-interference does not hold since the result of the program depends on some of the input variables. Therefore, the analysis is only able to conclude that at least one of the input variables may be used by the program, but it cannot determine which input variables are unused. ■

The example shows that non-interference is too strong a property in general. Of course, one could determine which input variables are unused by running multiple instances of the non-interference analysis Λ_{F}, each one of them classifying a single different input variable as H and all other variables as L. However, this becomes cumbersome in a data science application where a program reads and manipulates a large amount of input data.

Moreover, we emphasize that our input data usage property is more general than (possibilistic) non-interference since it also considers non-termination. We are not aware of any work on termination-sensitive possibilistic non-interference.

Example 8. Let us modify the program P shown in Example 7 as follows:

```
1 passing = True
2 while not english: english = False
```

In this case, since the loop at line 2 does not modify the output variable `passing`, the non-interference analysis Λ_{F} will leave the initial set of dependency constraints $\{L \rightsquigarrow \texttt{passing}\}$ unchanged, meaning that the result of the program does not depend on any of its input variables. However, the input variable `english` is used since its value influences the outcome of the program: the program terminates if `english` is true, and does not terminate otherwise. ■

The example demonstrates that the analysis is *unsound* for a non-terminating program.[2] We show that the non-interference analysis Λ_{F} is sound for proving that a program does not use any of its input variables, *only if the program is terminating*. We define the concretization function $\gamma_{\mathrm{F}} \colon \mathcal{P}(\mathrm{F}) \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$:

$$\gamma_{\mathrm{F}}(S) \stackrel{\text{def}}{=} \{R \in \mathcal{P}(\Sigma \times \Sigma) \mid \alpha_{\mathrm{F}}(R) \sqsubseteq_{\mathrm{F}} S\} \tag{17}$$

The abstraction function $\alpha_{\mathrm{F}} \colon \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma)) \to \mathcal{P}(\mathrm{F})$ maps each relation R between states of a program to the corresponding set of dependency constraints: $\alpha_{\mathrm{F}}(R) \stackrel{\text{def}}{=} \{L \rightsquigarrow x \mid x \in \mathrm{X}_L \wedge \forall i \in \mathrm{X}_H \colon \textsc{unused}_{i,x}(R)\}$, where $\textsc{unused}_{i,x}$ is the relational abstraction of $\textsc{unused}_i$ (cf. Eq. 3) in which we compare only the result stored in the variable x (i.e., we compare $\sigma[\omega](o)$ and $\sigma'[\omega](o)$, instead of $\sigma[\omega]$ and $\sigma'[\omega]$ as in Eq. 3).

Theorem 5. *A terminating program does not use any of its input variables if the image via $\gamma_{\rightsquigarrow} \circ \gamma_{\mathrm{F}}$ of its non-interference abstraction Λ_{F} is a subset of $\mathcal{N}$:*

$$\gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})) \subseteq \mathcal{N} \Rightarrow P \models \mathcal{N}$$

Proof. Let us assume that $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})) \subseteq \mathcal{N}$. By definition of γ_{F} (cf. Eq. 17), since the program is terminating, we have that $\Lambda_{\rightsquigarrow} \subseteq \gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})$ and, by monotonicity of the concretization function $\gamma_{\rightsquigarrow}$ (cf. Eq. 11), we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}}))$. Thus, since $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})) \subseteq \mathcal{N}$, we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}$. The conclusion follows from Theorem 4. □

Note that the termination of the program is necessary for the proof of Theorem 5. Indeed, for a non-terminating program, we have that $\Lambda_{\rightsquigarrow} \not\subseteq \gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})$ (since $\Lambda_{\rightsquigarrow}$ includes relational abstractions of infinite traces that are missing from $\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})$) and thus we cannot conclude the proof.

This result shows that the non-interference analysis Λ_{F} is an abstraction of the dependency semantics $\Lambda_{\rightsquigarrow}$ presented earlier. However, we remark that the same result applies to all other instances in this important class of analysis [5, 25, etc.], which are therefore subsumed by our framework.

9 Strongly Live Variable Abstraction

Strongly live variable analysis [20] is a variant of the classic live variable analysis [32] performed by compilers to determine, for each program point, which variables may be potentially used before they are assigned to. A variable is *strongly live* if it is used in an assignment to another strongly live variable, or if is used in a statement other than an assignment. Otherwise, a variable is considered *faint*.

[2] The case of a program using an input variable and then always diverging is not problematic because the analysis would be imprecise but still sound.

Strongly live variable analysis Λ_{X} is a *backward analysis* in the complete lattice $\langle \mathcal{P}(\mathrm{X}), \subseteq, \cup, \cap, \emptyset, \mathrm{X} \rangle$, where X is the set of all program variables. The transfer function $\Theta_{\mathrm{X}}[\![s]\!] \colon \mathcal{P}(\mathrm{X}) \to \mathcal{P}(\mathrm{X})$ for each statement s is defined as:

$$\begin{aligned}
\Theta_{\mathrm{X}}[\![\texttt{skip}]\!](S) &\stackrel{\text{def}}{=} S \\
\Theta_{\mathrm{X}}[\![x = e]\!](S) &\stackrel{\text{def}}{=} \begin{cases} (S \setminus \{x\}) \cup \textsc{vars}(e) & x \in S \\ S & \text{otherwise} \end{cases} \\
\Theta_{\mathrm{X}}[\![\texttt{if } b\texttt{: } s_1 \texttt{ else: } s_2]\!](S) &\stackrel{\text{def}}{=} \textsc{vars}(b) \cup \Theta_{\mathrm{X}}[\![s_1]\!](S) \cup \Theta_{\mathrm{X}}[\![s_2]\!](S) \\
\Theta_{\mathrm{X}}[\![\texttt{while } b\texttt{: } s]\!](S) &\stackrel{\text{def}}{=} \textsc{vars}(b) \cup \Theta_{\mathrm{X}}[\![s]\!](S) \\
\Theta_{\mathrm{X}}[\![s_1\ s_2]\!](S) &\stackrel{\text{def}}{=} \Theta_{\mathrm{X}}[\![s_1]\!] \circ \Theta_{\mathrm{X}}[\![s_2]\!](S)
\end{aligned}$$

where $\textsc{vars}(e)$ is the set of variables in the expression e. For input data usage, the initial set of strongly live variables contains the output variables of the program.

Example 9. Let us consider again the program P shown in Example 7. The strongly live variable analysis begins from the set {`passing`} containing the output variable `passing`. At line 3, the set of strongly live variables is {`math`, `bonus`} since `bonus` is used in an assignment to the strongly live variable `passing`, and `math` is used in the condition of the `if` statement. Finally, at line 1, the set of strongly live variables is {`english`, `math`, `bonus`} because `english` is used in the condition of the `if` statement at line 2. Thus, strongly live variable analysis is able to conclude that the input variable `science` is unused. However, it is not precise enough to determine that the variable `english` is also unused. ■

The imprecision of the analysis derives from the fact that it does not capture implicit flows of information precisely (cf. Sect. 8) but only over-approximates their presence. Thus, the analysis is unable to detect when a conditional statement, for instance, modifies only variables that have no impact on the outcome of a program; a situation likely to arise due to a programming error, as shown in the previous example. However, in virtue of this imprecise treatment of implicit flows, we can show that strongly live variable analysis is sound for input data usage, even for non-terminating programs.

We define the concretization function $\gamma_{\mathrm{X}} \colon \mathcal{P}(X) \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\perp))$ as:

$$\gamma_{\mathrm{X}}(S) \stackrel{\text{def}}{=} \{R \in \Sigma \times \Sigma_\perp \mid \forall i \in \mathrm{X} \setminus S \colon \textsc{unused}_i(R)\} \tag{18}$$

where we abuse notation and use $\textsc{unused}_i$ (cf. Eq. 3) to also denote its dependency abstraction (cf. Eq. 10). We now show that strongly live variable analysis is sound for proving that a program does not use the faint variables.

Theorem 6. *A program does not use a subset J of its input variables if the image via $\gamma_{\rightsquigarrow} \circ \gamma_{\mathrm{X}}$ of its strongly live variable abstraction Λ_{X} is a subset of $\mathcal{N}_J$:*

$$\gamma_{\rightsquigarrow}(\gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}})) \subseteq \mathcal{N}_J \Rightarrow P \models \mathcal{N}_J$$

Proof. Let us assume that $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}})) \subseteq \mathcal{N}_J$. By definition of γ_{X} (cf. Eq. 18), we have that $\Lambda_{\rightsquigarrow} \subseteq \gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}})$ and, by monotonicity of $\gamma_{\rightsquigarrow}$ (cf. Eq. 11), we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \gamma_{\rightsquigarrow}(\gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}}))$. Thus, since $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}})) \subseteq \mathcal{N}_J$, we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}_J$. The conclusion follows from Theorem 4. □

This result shows that also strongly live variable analysis is subsumed by our framework as it is an abstraction of the dependency semantics $\Lambda_{\rightsquigarrow}$.

10 Syntactic Dependency Abstractions

In the following, we derive a more precise data usage analysis based on *syntactic* dependencies between program variables. For simplicity, the analysis does not take program termination into account, but we discuss possible solutions at the end of the section. Due to space limitations, we only provide a terse description of the abstraction and refer to [36] for further details.

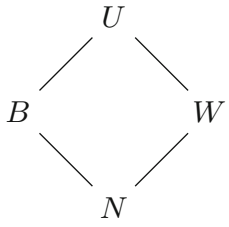


Fig. 5. Hasse diagram for the complete lattice $\langle \textsc{usage}, \sqsubseteq_{\textsc{usage}}, \sqcup_{\textsc{usage}}, \sqcap_{\textsc{usage}}, N, U \rangle$.

In order to capture implicit dependencies from variables appearing in boolean conditions of conditional and while statements, we track when the value of a variable is used or modified in a statement based on the level of nesting of the statement in other statements. More formally, each program variable maps to a value in the complete lattice shown in Fig. 5: the values U (*used*) and N (*not-used*) respectively denote that a variable may be used and is not used at the current nesting level; the values B (*below*) and W (*overwritten*) denote that a variable may be used at a lower nesting level, and the value W additionally indicates that the variable is modified at the current nesting level.

A variable is used (i.e., maps to U) if it is used in an assignment to another variable that is used in the current or a lower nesting level (i.e., a variable that maps to U or B). We define the operator $\textsc{assign}[\![x = e]\!]$ to compute the effect of an assignment on a map $m \colon \mathrm{X} \to \textsc{usage}$, where X is the set of all variables:

$$\textsc{assign}[\![x = e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} W & y = x \wedge y \notin \textsc{vars}(e) \wedge m(x) \in \{U, B\} \\ U & y \in \textsc{vars}(e) \wedge m(x) \in \{U, B\} \\ m(y) & \text{otherwise} \end{cases} \tag{19}$$

The assigned variable is overwritten (i.e., maps to W), unless it is used in e.

Another reason for a variable to be used is if it appears in the boolean condition e of a statement that uses another variable or modifies another used variable (i.e., there exists a variable x that maps to U or W):

$$\textsc{filter}[\![e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} U & y \in \textsc{vars}(e) \wedge \exists x \in \mathrm{X}\colon m(x) \in \{U, W\} \\ m(y) & \text{otherwise} \end{cases} \tag{20}$$

We maintain a *stack* of these maps that grows or shrinks based on the level of nesting of the currently analyzed statement. More formally, a stack is a tuple $\langle m_0, m_1, \ldots, m_k \rangle$ of mutable length k, where each element $m_0, m_1, \ldots, m_k$ is a map from X to USAGE. In the following, we use Q to denote the set of all stacks, and we abuse notation by writing $\textsc{assign}[\![x = e]\!]$ and $\textsc{filter}[\![e]\!]$ to also denote the corresponding operators on stacks:

$$\textsc{assign}[\![x = e]\!](\langle m_0, m_1, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \langle \textsc{assign}[\![x = e]\!](m_0), m_1, \ldots, m_k \rangle$$
$$\textsc{filter}[\![e]\!](\langle m_0, m_1, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \langle \textsc{filter}[\![e]\!](m_0), m_1, \ldots, m_k \rangle$$

The operator PUSH duplicates the map at the top of the stack and modifies the copy using the operator INC, to account for an increased nesting level:

$$\begin{aligned} \textsc{push}(\langle m_0, m_1, \ldots, m_k \rangle) &\stackrel{\text{def}}{=} \langle \textsc{inc}(m_0), m_0, m_1, \ldots, m_k \rangle \\ \textsc{inc}(m) &\stackrel{\text{def}}{=} \lambda y. \begin{cases} B & m(y) \in \{U\} \\ N & m(y) \in \{W\} \\ m(y) & \text{otherwise} \end{cases} \end{aligned} \tag{21}$$

A used variable (i.e., mapping to U) becomes used below (i.e., now maps to B), and a modified variable (i.e., mapping to W) becomes unused (i.e., now maps to N). The dual operator POP combines the two maps at the top of the stack:

$$\begin{aligned} \textsc{pop}(\langle m_0, m_1, \ldots, m_k \rangle) &\stackrel{\text{def}}{=} \langle \textsc{dec}(m_0, m_1), \ldots, m_k \rangle \\ \textsc{dec}(m, k) &\stackrel{\text{def}}{=} \lambda y. \begin{cases} k(y) & m(y) \in \{B, N\} \\ m(y) & \text{otherwise} \end{cases} \end{aligned} \tag{22}$$

where the DEC operator restores the value a variable y mapped to before increasing the nesting level (i.e., $k(y)$) if it has not changed since (i.e., if the variable still maps to B or N), and otherwise retains the new value y maps to.

We can now define the data usage analysis Λ_{Q}, which is a *backward analysis* on the lattice $\langle \mathrm{Q}, \sqsubseteq_{\mathrm{Q}}, \sqcup_{\mathrm{Q}} \rangle$. The partial order $\sqsubseteq_{\mathrm{Q}}$ and the least upper bound $\sqcup_{\mathrm{Q}}$ are the pointwise lifting, for each element of the stack, of the partial order and least upper bound between maps from X to USAGE (which in turn are the pointwise lifting of the partial order $\sqsubseteq_{\textsc{usage}}$ and least upper bound $\sqcup_{\textsc{usage}}$ of the USAGE lattice, cf. Fig. 5). We define the transfer function $\Theta_{\mathrm{Q}}[\![s]\!]\colon \mathrm{Q} \to \mathrm{Q}$ for each statement s in our simple programming language as follows:

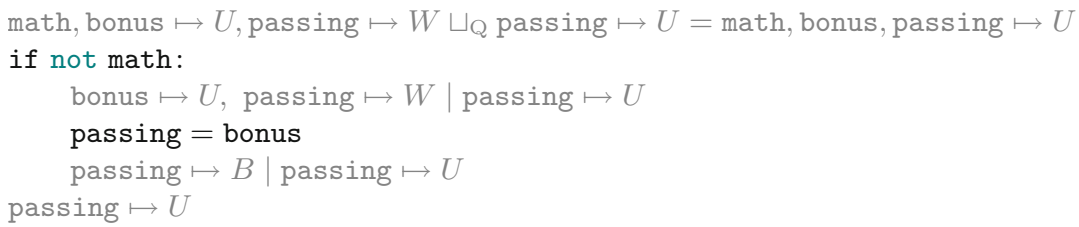


Fig. 6. Data usage analysis of the last statement of the program shown in Example 7. Stack elements are separated by | and, for brevity, variables mapping to N are omitted.

$$
\begin{aligned}
\Theta_{\mathrm{Q}}[\![\texttt{skip}]\!](q) &\stackrel{\text{def}}{=} q \\
\Theta_{\mathrm{Q}}[\![x = e]\!](q) &\stackrel{\text{def}}{=} \textsc{assign}[\![x = e]\!](q) \\
\Theta_{\mathrm{Q}}[\![\texttt{if } b\texttt{: } s_1 \texttt{ else: } s_2]\!](q) &\stackrel{\text{def}}{=} \textsc{pop} \circ \textsc{filter}[\![b]\!] \circ \Theta_{\mathrm{Q}}[\![s_1]\!] \circ \textsc{push}(q) \\
&\quad \sqcup_{\mathrm{Q}} \textsc{pop} \circ \textsc{filter}[\![b]\!] \circ \Theta_{\mathrm{Q}}[\![s_2]\!] \circ \textsc{push}(q) \\
\Theta_{\mathrm{Q}}[\![\texttt{while } b\texttt{: } s]\!](q) &\stackrel{\text{def}}{=} \mathrm{lfp}_t^{\sqsubseteq_{\mathrm{Q}}}\, \Theta_{\mathrm{Q}}[\![\texttt{if } b\texttt{: } s \texttt{ else: skip}]\!] \\
\Theta_{\mathrm{Q}}[\![s_1\ s_2]\!](q) &\stackrel{\text{def}}{=} \Theta_{\mathrm{Q}}[\![s_1]\!] \circ \Theta_{\mathrm{Q}}[\![s_2]\!](q)
\end{aligned}
$$

The initial stack contains a single map, in which the output variables map to the value U, and all other variables map to N. We exemplify the analysis below.

Example 10. Let us consider again the program P shown in Example 7. The initial stack begins with a single map m, in which the output variable `passing` maps to U and all other variables map to N.

At line 4, before analyzing the body of the conditional statement, a modified copy of m is pushed onto the stack: this copy maps `passing` to B, meaning that `passing` is only used in a lower nesting level, and all other variables still map to N (cf. Eq. 21). As a result of the assignment (cf. Eq. 19), `passing` is overwritten (i.e., maps to W), and bonus is used (i.e., maps to U). Since the body of the conditional statement modifies a used variable and uses another variable, the analysis of its boolean condition makes `math` used as well (cf. Eq. 20). Finally, the maps at the top of the stack are merged and the result maps `math`, `bonus`, and `passing` to U, and all other variables to N (cf. Eq. 22). The analysis is visualized in Fig. 6.

The stack remains unchanged at line 3 and line 2, since the statement at line 3 is identical to line 4 and the body of the conditional statement at line 2 does not modify any used variable and does not use any other variable. Finally, at line 1 the variable `passing` is modified (i.e., it now maps to W), while `math` and `bonus` remain used (i.e., they map to U). Thus, the analysis is precise enough to conclude that the input variables `english` and `science` are unused. ■

Note that, similarly to the non-interference analysis presented in Sect. 8, the data usage analysis Λ_{Q} does not consider non-termination. Indeed, for the program shown in Example 8, the analysis does not capture that the input variable

`english` is used, even though the termination of the program depends on its value. We define the concretization function $\gamma_{\mathrm{Q}} \colon \mathrm{Q} \to \mathcal{P}\left(\mathcal{P}\left(\Sigma \times \Sigma\right)\right)$ as:

$$\gamma_{\mathrm{Q}}(\langle m_0, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \{R \in \Sigma \times \Sigma \mid \forall i \in \mathrm{X} \colon m_0(i) \in \{N\} \Rightarrow \textsc{unused}_i(R)\} \tag{23}$$

where again we write $\textsc{unused}_i$ (cf. Eq. 3) to also denote its dependency abstraction. We now show that Λ_{Q} is sound for proving that a program does not use a subset of its input variables, *if the program is terminating.*

Theorem 7. *A terminating program does not use a subset J of its input variables if the image via $\gamma_{\rightsquigarrow} \circ \gamma_{\mathrm{Q}}$ of its abstraction Λ_{Q} is a subset of $\mathcal{N}_J$:*

$$\gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})) \subseteq \mathcal{N}_J \Rightarrow P \models \mathcal{N}_J$$

Proof. Let us assume that $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})) \subseteq \mathcal{N}_J$. Since the program is terminating, we have that $\Lambda_{\rightsquigarrow} \subseteq \gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})$, by definition of the concretization function γ_{Q} (cf. Eq. 23). Then, by monotonicity of $\gamma_{\rightsquigarrow}$ (cf. Eq. 11), we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}}))$. Thus, since $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})) \subseteq \mathcal{N}_J$, we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}_J$. The conclusion follows from Theorem 4. □

In order to take termination into account, one could map each variable appearing in the guard of a loop to the value U. Alternatively, one could run a termination analysis [3,33,34], along with the data usage analysis, and only map to U variables appearing in the loop guard of a possibly non-terminating loop.

11 Piecewise Abstractions

The static analyses presented so far can be used only to detect unused data stored in program variables. However, realistic data science applications read and manipulate data organized in data structures such as arrays, lists, and dictionaries. In the following, we demonstrate that having expressed the analyses as abstract domains allows us to easily lift the analyses to such a scenario. In particular, to detect unused chunks of the input data, we combine the more precise data usage analysis presented in the previous section with the array content abstraction proposed by Cousot et al. [16]. Due to space limitations, we provide only an informal description of the resulting abstract domain and refer to [36] for further details and examples. The analyses presented in earlier sections can be similarly combined with the array abstraction for the same purpose.

We extend our small programming language introduced in Sect. 7 with integer variables, arithmetic and boolean comparison expressions, and arrays:

$$\begin{aligned} e &::= \cdots \mid a[e] \mid \texttt{len}(a) \mid e \ \oplus \ e \mid e \ \bowtie \ e && \text{(expressions)} \\ s &::= \cdots \mid a[e] = e && \text{(statements)} \end{aligned}$$

where $\oplus$ and $\bowtie$ respectively range over arithmetic and boolean comparison operators, a ranges over array variables, and $\texttt{len}(a)$ denotes the length of a.

Piecewise Array Abstraction. The array abstraction [16] divides an array into consecutive segments, each segment being a uniform abstraction of the array content in that segment. The bounds of the segments are specified by sets of side-effect free expressions restricted to a canonical normal form, all having the same (concrete) value. The abstraction is parametric in the choice of the abstract domains used to manipulate sets of expressions and to represent the array content within each segment. For our analysis, we use the octagon abstract domain [31] for the expressions, and the USAGE lattice presented in the previous section (cf. Fig. 5) for the segments. Thus, an array a is abstracted, for instance, as $\{0, i\}\, N\, \{j+1\}?\; U\, \{\texttt{len}(a)\}$, where the symbol ? indicates that the segment $\{0, i\}\, N\, \{j+1\}$ might be empty. The abstraction indicates that all array elements (if any) from index i (which is equal to zero) to index j (the bound $j+1$ is exclusive) are unused, and all elements from $j+1$ to $\texttt{len}(a)-1$ may be used. Let A be the set of all such array abstractions. The initial segmentation of an array $a \in \mathrm{A}$ is a single segment with unused content (i.e., $\{0\}\, N\, \{\texttt{len}(a)\}?$).

For our analysis, we augment the array abstraction with new backward assignment and filter operators. The operators $\textsc{assign}_{\mathrm{A}}[\![a[i] = e]\!]$ and $\textsc{filter}_{\mathrm{A}}[\![e]\!]$ split and fill segments to take into account assignments and accesses to array elements that influence the program outcome. For instance, an assignment to $a[i]$ with an expression containing a used variable modifies the segmentation $\{0\}\, N\, \{\texttt{len}(a)\}?$ into $\{0\}\, N\, \{i\}?\; U\, \{i+1\}\, N\, \{\texttt{len}(a)\}?$, which indicates that the array element at index i is used by the program. An access $a[i]$ in a boolean condition guarding a statement that uses or modifies another used variables is handled analogously. Instead, the operator $\textsc{assign}_{\mathrm{A}}[\![x = e]\!]$ modifies the segmentation of an array by replacing each occurrence of the assigned variable x with the canonical normal form of the expression e. For instance, an assignment $i = i+1$ modifies the segmentation $\{0\}\, N\, \{i\}?\; U\, \{i+1\}\, N\, \{\texttt{len}(a)\}?$ into $\{0\}\, N\, \{i+1\}?\; U\, \{i+2\}\, N\, \{\texttt{len}(a)\}?$. If e cannot be precisely put into a canonical normal form, the operator replaces the assigned variable with an approximation of e as an integer interval [13] computed using the underlying numerical domain, and possibly merges segments together as a result of the approximation. For instance, a non-linear assignment $i = i * j$ approximated as $i = [0, 1]$ modifies the segmentation $\{0\}\, N\, \{i\}?\; U\, \{i+1\}\, N\, \{\texttt{len}(a)\}?$ into $\{0\}\, U\, \{2\}\, N\, \{\texttt{len}(a)\}?$, which loses the information that the initial segment of the array is unused.

When merging control flows, segmentations are compared or joined by means of a *unification algorithm* [16], which finds the coarsest common refinement of both segmentations. Then, the comparison $\sqsubseteq_{\mathrm{A}}$ or the join $\sqcup_{\mathrm{A}}$ is performed pointwise for each segment using the corresponding operators of the underlying abstract domain chosen to abstract the array content. For our analysis, we adapt and refine the originally proposed unification algorithm to take into account the knowledge of the numerical domain chosen to abstract the segment bounds. We refer to [36] for further details. A widening $\triangledown_{\mathrm{A}}$ limits the number of segments to enforce termination of the analysis.

Piecewise Data Usage Analysis. We can now map each scalar variable to an element of the USAGE lattice and each array variable to an array segmentation

```
1 failed = 0
2 i = 1                                                    # 1 should be 0
3 while i < len(grades):
4     if grades[i] < 4: failed = failed + 1
5     i = i + 1
6 passing = 2 * failed < len(grades)
```

Fig. 7. Another program to check if a student has passed a number of exams based on their grades stored in the array `grades`. The programmer has made a mistake at line 2 that causes the program to ignore the grade stored at index 0 in `grades`.

```
grades ↦ {0} N {i}? U {i + 1}? U {len(grades)}?
while i < len(grades):
    grades ↦ {0} N {i}? U {i + 1}? B {i + 2}? B {len(grades)}? | ...
    if grades[i] < 4:
        grades ↦ {0} N {i + 1}? B {i + 2}? B {len(grades)}? | ··· | ...
        failed = failed + 1
        grades ↦ {0} N {i + 1}? B {i + 2}? B {len(grades)}? | ··· | ...
    grades ↦ {0} N {i + 1}? B {i + 2}? B {len(grades)}? | ...
    i = i + 1
    grades ↦ {0} N {i}? B {i + 1}? B {len(grades)}? | ...
grades ↦ {0} N {len(grades)}?
```

Fig. 8. Data usage analysis of the loop statement of the program shown in Example 11. Stack elements are separated by | and, for brevity, only array variables are shown.

in A, and use the data usage analysis Λ_{Q} presented in the previous section to identify unused input data stored in variables and portions of arrays.

Example 11. Let us consider the program shown in Fig. 7 where the array variable `grades` and the variable `passing` are the input and output variables, respectively. The initial stack contains a single map in which `passing` maps to U, all other scalar variables map to N, and `grades` maps to $\{0\}\,N\,\{\texttt{len(grades)}\}?$, indicating that all elements of the array (if any) are unused.

At line 6, the assignment modifies the variable `passing` (i.e., `passing` now maps to W) and uses the variable `failed` (i.e., `failed` now maps to U), while every other variable remains unchanged.

The result of the analysis of the loop statement at line 3 is shown in Fig. 8. The analysis of the loop begins by pushing (cf. Eq. 21) a map onto the stack in which `passing` becomes unused (i.e., maps to N) and `failed` is used only in a lower nesting level (i.e., maps to B), and every other variable still remains unchanged. At the first iteration of the analysis of the loop body, the assignment at line 4 uses `failed` and thus the access `grades[i]` at line 3 creates a used segment in the segmentation for `grades`, which becomes $\{0\}\,N\,\{\texttt{i}\}?\ U\,\{\texttt{i}+1\}\ \ N\,\{\texttt{len(grades)}\}?$. At the second iteration, the PUSH operator turns the used segment $\{\texttt{i}\}\,U\,\{\texttt{i}+1\}$ into $\{\texttt{i}\}\,B\,\{\texttt{i}+1\}$, and the assignment to `i` modifies the segment into $\{\texttt{i}+1\}\,B\,\{\texttt{i}+2\}$ (while the segmentation in the second stack element becomes

$\{0\}\, N\, \{\texttt{i}+1\}?\ U\, \{\texttt{i}+2\}\ \ N\, \{\texttt{len(grades)}\}?$). Then, the access to the array at line 3 creates again a used segment $\{\texttt{i}\}\, U\, \{\texttt{i}+1\}$ (in the first segmentation) and the analysis continues with the result of the POP operator (cf. Eq. 22): $\{0\}\, N\, \{\texttt{i}\}?\ U\, \{\texttt{i}+1\}?\ U\, \{\texttt{i}+2\}?\ N\, \{\texttt{len(grades)}\}?$. After widening, the last two segments are merged into a single segment, and the analysis of the loop terminates with $\{0\}\, N\, \{\texttt{i}\}?\ U\, \{\texttt{i}+1\}?\ U\, \{\texttt{len(grades)}\}?$.

Finally, the analysis of the assignment at line 2 produces the segmentation $\{0\}\, N\, \{1\}?\ U\, \{2\}?\ U\, \{\texttt{len(grades)}\}?$, which correctly indicates that the first element of the array `grades` (if any) is unused by the program. ■

Implementation. The analyses presented in this and in the previous section are implemented in the prototype static analyzer LYRA and are available online[3].

The implementation is in PYTHON and, at the time of writing, accepts programs written in a limited subset of PYTHON without user-defined classes. A type inference is run before the analysis of a program. The analysis is performed backwards on the control flow graph of the program with a standard worklist algorithm [32], using widening at loop heads to enforce termination.

12 Related Work

The most directly relevant work has been discussed throughout the paper. The non-interference analysis proposed by Assaf et al. [6] (cf. Sect. 8) is similar to the logic of Amtoft and Banerjee [5] and the type system of Hunt and Sands [25]. The data usage analysis proposed in Sect. 10 is similar to dependency analyses used for program slicing [37] (e.g., [24]). Both analyses as well as strongly live variable analysis (cf. Sect. 9) are based on the *syntactic* presence of a variable in the definition of another variable. To overcome this limitation, one should look further for *semantic* dependencies between *values* of program variables. In this direction, Giacobazzi, Mastroeni, and others [19,22,29] have proposed the notion of *abstract dependency*. However, note that an analysis based on abstract dependencies would over-approximate the subset of the input variables that are unused by a program. Indeed, the absence of an abstract dependency between variables (e.g., a dependency between the parity of the variables [19,29]) does not imply the absence of a (concrete) dependency between the variables (i.e., a dependency between the values of the variables). Thus, such an analysis could not be used to prove that a program *does not use* a subset of its input variables, but would be used to prove that a program *uses* a subset of its input variables.

Semantics formulations using *sets of sets of traces* have already been proposed in the literature [6,28]. Mastroeni and Pasqua [28] lift the hierarchy of semantics developed by Cousot [12] to sets of sets of traces to obtain a hierarchy of semantics suitable for verifying general program properties (i.e., properties that are not subset-closed, cf. Sect. 7). However, *none* of the semantics that they proposed is suitable for input data usage: all semantics in the hierarchy are abstractions of a semantics that contains sets with both finite and infinite traces

[3] http://www.pm.inf.ethz.ch/research/lyra.html.

and thus, unlike our outcome semantics (cf. Sect. 5), cannot be used to reason about terminating and non-terminating outcomes of a program. Similarly, as observed in [28], the semantics proposed by Assaf et al. [6] can be used to verify only subset-closed properties. Thus, it cannot be used for input data usage.

Finally, to the best of our knowledge, our work is the first to aim at detecting programming errors in data science code using static analysis. Closely related are [7,10] which, however, focus on spreadsheet applications and target errors in the data rather than the code that analyzes it. Recent work [2] proposes an approach to repair *bias* in data science code. We believe that our work can be applied in this context to prove absence of bias, e.g., by showing that a program does not use gender information to decide whether to hire a person.

13 Conclusion and Future Work

In this paper, we have proposed an abstract interpretation framework to automatically detect input data that remains unused by a program. Additionally, we have shown that existing static analyses based on dependencies are subsumed by our unifying framework and can be used, with varying degrees of precision, for proving that a program does not use some of its input data. Finally, we have proposed a data usage analysis for more realistic data science applications that store input data in compound data structures such as arrays or lists.

As part of our future work, we plan to use our framework to guide the design of new, more precise static analyses for data usage. We also want to explore the complementary direction of proving that a program *uses* its input data by developing an analysis based on abstract dependencies [19,22,29] between program variables, as discussed above. Additionally, we plan to investigate other applications of our work such as provenance or lineage analysis [9] as well as proving absence of algorithmic bias [2]. Finally, we want to study other programming errors related to data usage such as accidental data duplication.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: POPL, pp. 147–160 (1999)
2. Albarghouthi, A., D'Antoni, L., Drews, S.: Repairing decision-making programs under uncertainty. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 181–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_9
3. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_8
4. Alpern, B., Schneider, F.B.: Defining Liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
5. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 100–115. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27864-1_10

6. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: POPL, pp. 874–887 (2017)
7. Barowy, D.W., Gochev, D., Berger, E.D.: CheckCell: data debugging for spreadsheets. In: OOPSLA, pp. 507–523 (2014)
8. Binkley, D., Gallagher, K.B.: Program slicing. Adv. Comput. **43**, 1–50 (1996)
9. Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. Math. Struct. Comput. Sci. **21**(6), 1301–1337 (2011)
10. Cheng, T., Rival, X.: Static analysis of spreadsheet applications for type-unsafe operations detection. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 26–52. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_2
11. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
12. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoret. Comput. Sci. **277**(1–2), 47–103 (2002)
13. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Symposium on Programming, pp. 106–130 (1976)
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
15. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
16. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL, pp. 105–118 (2011)
17. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
18. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
19. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: parameterizing non-interference by abstract interpretation. In POPL, pp. 186–197 (2004)
20. Giegerich, R., Möncke, U., Wilhelm, R.: Invariance of approximate semantics with respect to program transformations. In: Brauer, W. (ed.) GI - 11. Jahrestagung. Informatik-Fachberichte, vol. 50. Springer, Heidelberg (1981). https://doi.org/10.1007/978-3-662-01089-1_1
21. Goguen, J.A., Meseguer, J.: Security policies and security models. In: S & P, pp. 11–20 (1982)
22. Halder, R., Cortesi, A.: Abstract program slicing on dependence condition graphs. Sci. Comput. Program. **78**(9), 1240–1263 (2013)
23. Herndon, T., Ash, M., Pollin, R.: Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff. Camb. J. Econ. **38**(2), 257–279 (2014)
24. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990)
25. Hunt, S., Sands, D.: On flow-sensitive security types. In: POPL, pp. 79–90 (2006)
26. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2), 125–143 (1977)
27. Leveson, N.G., Turner, C.S.: Investigation of the Therac-25 accidents. IEEE Comput. **26**(7), 18–41 (1993)
28. Mastroeni, I., Pasqua, M.: Hyperhierarchy of semantics - a formal framework for hyperproperties verification. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 232–252. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_12

29. Mastroeni, I., Zanardini, D.: Abstract program slicing: an abstract interpretation-based approach to program slicing. ACM Trans. Comput. Log. **18**(1), 7:1–7:58 (2017)
30. Mencinger, J., Aristovnik, A., Verbic, M.: The impact of growing public debt on economic growth in the European Union. Amfiteatru Econ. **16**(35), 403–414 (2014)
31. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006)
32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
33. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_20
34. Urban, C.: The abstract domain of segmented ranking functions. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 43–62. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_5
35. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996)
36. Wehrli, S.: Static program analysis of data usage properties. Master's thesis, ETH Zurich, Zurich, Switzerland (2017)
37. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984)

[c17] **CU**, Maria Christakis, Valentin Wüstholz, Fuyuan Zhang

Perfectly Parallel Fairness Certification of Neural Networks

In ACM on Programming Languages (PACMPL), Conference on Object-Oriented Programming Systems, Languages, and Applications 2020 (OOPSLA 2020)

https://inria.hal.science/hal-03091870

Perfectly Parallel Fairness Certification of Neural Networks

CATERINA URBAN, INRIA and DIENS, École Normale Supérieure, CNRS, PSL University, France
MARIA CHRISTAKIS, MPI-SWS, Germany
VALENTIN WÜSTHOLZ, ConsenSys, Germany
FUYUAN ZHANG, MPI-SWS, Germany

Recently, there is growing concern that machine-learned software, which currently assists or even automates decision making, reproduces, and in the worst case reinforces, bias present in the training data. The development of tools and techniques for certifying fairness of this software or describing its biases is, therefore, critical. In this paper, we propose a *perfectly parallel* static analysis for certifying *fairness* of feed-forward neural networks used for classification of tabular data. When certification succeeds, our approach provides definite guarantees, otherwise, it describes and quantifies the biased input space regions. We design the analysis to be *sound*, in practice also *exact*, and configurable in terms of scalability and precision, thereby enabling *pay-as-you-go certification*. We implement our approach in an open-source tool called LIBRA and demonstrate its effectiveness on neural networks trained on popular datasets.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Program analysis**; **Abstraction**; • **Computing methodologies** → *Neural networks*; • **Social and professional topics** → **Computing / technology policy**.

Additional Key Words and Phrases: Fairness, Neural Networks, Abstract Interpretation, Static Analysis

ACM Reference Format:
Caterina Urban, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Perfectly Parallel Fairness Certification of Neural Networks. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 185 (November 2020), 30 pages. https://doi.org/10.1145/3428253

1 INTRODUCTION

Due to the tremendous advances in machine learning and the vast amounts of available data, machine-learned software has an ever-increasing important role in assisting or even autonomously making decisions that impact our lives. We are already witnessing the wide adoption and societal impact of such software in criminal justice, health care, and social welfare, to name a few examples. It is not far-fetched to imagine a future where most of the decision-making is automated.

However, several studies have recently raised concerns about the fairness of such systems. For instance, consider a commercial recidivism-risk assessment algorithm that was found racially biased [Larson et al. 2016]. Similarly, a commercial algorithm that is widely used in the U.S. health care system falsely determined that Black patients were healthier than other equally sick patients by using health costs to represent health needs [Obermeyer et al. 2019]. There is also empirical evidence of gender bias in image searches, for instance, there are fewer results depicting women when searching for certain occupations, such as CEO [Kay et al. 2015]. Commercial facial recognition

Authors' addresses: Caterina Urban, INRIA and DIENS, École Normale Supérieure, CNRS, PSL University, Paris, France, caterina.urban@inria.fr; Maria Christakis, MPI-SWS, Germany, maria@mpi-sws.org; Valentin Wüstholz, ConsenSys, Germany, valentin.wustholz@consensys.net; Fuyuan Zhang, MPI-SWS, Germany, fuyuan@mpi-sws.org.

2475-1421/2020/11-ART185
https://doi.org/10.1145/3428253

algorithms, which are increasingly used in law enforcement, are less effective for women and darker skin types [Buolamwini and Gebru 2018].

In other words, machine-learned software may reproduce, or even reinforce, bias that is directly or indirectly present in the training data. This awareness will certainly lead to regulations and strict audits in the future. It is, therefore, critical to develop tools and techniques for certifying fairness of machine-learned software or understanding the circumstances of its biased behavior.

Feed-Forward Neural Networks. We make a step forward in meeting these needs by designing a novel static analysis framework for certifying fairness of *feed-forward neural networks* used *for classification of tabular data* (e.g., stored in Excel files or relational databases).

While, in general, machine learning approaches such as random forests or gradient boosting are preferred over neural networks for analyzing tabular data, for some datasets, feed-forward neural networks provide better accuracy and flexibility at the cost of putting slightly more effort into hyperparameter optimizations. This is the case, for instance, for very large datasets (i.e., with billions of rows) or data that comes in batches over time. As these kinds of datasets are becoming more and more common, so are neural network-based decision-making software systems.

The other challenge is that neural networks are harder to interpret than such machine learning approaches and, thus, it becomes harder to ensure that neural network-based decision-making software is doing what intended. This is precisely what our work aims to address.

Dependency Fairness. Our static analysis framework is designed for certifying *dependency fairness* [Galhotra et al. 2017][1] of feed-forward neural networks used for classification tasks. Specifically, given a choice (e.g., driven by a causal model [Pearl 2009] or a correlation analysis) of input features that are considered (directly or indirectly) sensitive to bias, *a neural network is fair if the output classification is not affected by different values of the chosen features.* We chose this definition over other fairness notions because it does not require an oracle and it is specifically designed to be testable and, thus, also amenable to static analysis. It is also a stronger notion than group-based fairness notions (see Section 12 for a more in-depth comparison).

Of course, an obvious way to avoid dependencies between the classification and the sensitive features is to entirely remove the sensitive features from the training data [Grgić-Hlača et al. 2016]. However, this in general does not work for three main reasons. First, neural networks learn from latent variables (e.g., [Lum and Isaac 2016; Udeshi et al. 2018]). For instance, a credit-screening algorithm that does not take race (or gender) as an input might still be biased with respect to it, say, by using the ZIP code of applicants as proxy for race (or their first name as proxy for gender). Therefore, simply removing a sensitive feature does not necessarily free the training data or the corresponding trained neural network from bias. Second, the training data is only a relatively small sample of the entire input space, on portions of which the neural network might end up being biased even if trained with fair data. For example, if women are under-represented in the training data, this leaves freedom to the training process and thus the resulting classifier might end up being biased against them. Third, the information provided by a sensitive feature might be necessary for business necessity [Barocas and Selbst 2016], for instance, to introduce intended bias in a certain input region. Assume a credit-screening model that should discriminate with respect to age only above a particular threshold. Above this age threshold, the higher the requested credit amount, the lower the chances of receiving it. In such cases, removing the sensitive feature is not even possible.

[1]Note that, Galhotra et al. [Galhotra et al. 2017] use the term *causal fairness* instead of *dependency fairness*. We renamed it to avoid confusion since this fairness notion does not presume a given *causal model* [Pearl 2009] but instead looks at whether the classification *depends* on the value of the sensitive features.

Our Approach. Static analysis of neural networks is still in its infancy. Most of the work in the area focuses on certifying local robustness of neural networks against adversarial examples [Gehr et al. 2018; Huang et al. 2017; Singh et al. 2019, etc.] or proving functional properties of neural networks [Katz et al. 2017; Wang et al. 2018, etc.]. On the other hand, dependency fairness is a *global* property, relative to the entire input space, instead of only those inputs within a particular distance metric or that satisfy a given functional specification. It is thus much harder to verify. The approach that we propose in this paper brings us closer to the aspiration for a practical general verification approach of global neural network properties.

Our approach certifies dependency fairness of feed-forward neural networks by employing a combination of a forward and a backward static analysis. On a high level, the forward pass aims to reduce the overall analysis effort. At its core, it divides the input space of the network into independent partitions. The backward analysis then attempts to certify fairness of the classification within each partition (in a *perfectly parallel* fashion) with respect to a chosen (set of) feature(s), which may be directly or indirectly sensitive, for instance, race or ZIP code. In the end, our approach reports for which regions of the input space the neural network is certified to be fair and for which there is bias. Note that we do not necessarily need to analyze the entire input space; our technique is also able to answer specific bias queries about a fraction of the input space, e.g., are Hispanics over 45 years old discriminated against with respect to gender?

The scalability-vs-precision tradeoff of our approach is configurable. Partitions that do not satisfy the given configuration are excluded from the analysis and may be resumed later, with a more flexible configuration. This enables usage scenarios in which our approach adapts to the available resources, e.g., time or CPUs, and is run incrementally. In other words, we designed a *pay-as-you-go certification* approach that the more resources it is given, the larger the region of the input space it is able to analyze.

Related Work. In the literature, related work on determining fairness of machine-learning models has focused on providing probabilistic guarantees [Bastani et al. 2019]. In contrast, our approach gives definite guarantees for those input partitions that satisfy the analysis configuration. Similarly to our approach, there is work that also aims to provide definite guarantees [Albarghouthi et al. 2017b] (although for different fairness criteria). However, it has been shown to scale only up to neural networks with two hidden neurons. We demonstrate that our approach can effectively analyze networks with hundreds (or, in some cases, even thousands) of hidden neurons. Ours is also the first approach in this area that is configurable in terms of scalability and precision.

Contributions. We make the following contributions:

(1) We propose a novel perfectly parallel static analysis approach for certifying dependency fairness of feed-forward neural networks used for classification of tabular data. If certification fails, our approach can describe and quantify the biased input space region(s).
(2) We rigorously formalize our approach and show that it is sound and, in practice, exact for the analyzed regions of the input space.
(3) We discuss the configurable scalability-vs-precision tradeoff of our approach that enables pay-as-you-go certification.
(4) We implement our approach in an open-source tool called LIBRA and extensively evaluate it on neural networks trained with popular datasets. We show the effectiveness of our approach in detecting injected bias and answering bias queries. We also experiment with the precision and scalability of the analysis and discuss the tradeoffs.

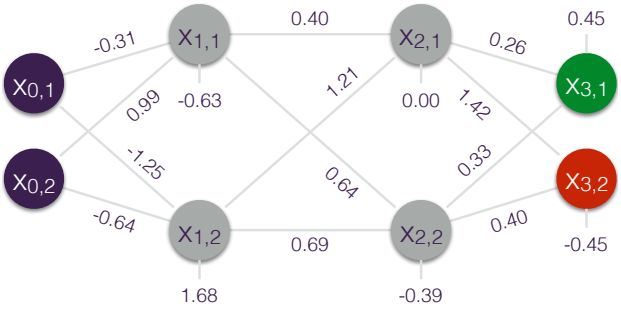


Fig. 1. Small, constructed example of trained feed-forward neural network for credit approval.

2 OVERVIEW

In this section, we give an overview of our approach using a small constructed example, which is shown in Figure 1.

Example. The figure depicts a feed-forward neural network for credit approval. There are two inputs $x_{0,1}$ and $x_{0,2}$ (shown in purple). Input $x_{0,1}$ denotes the requested credit amount and $x_{0,2}$ denotes age. Both inputs have continuous values in the range $[0, 1]$. Output $x_{3,2}$ (shown in green) denotes that the credit request is approved, whereas $x_{3,1}$ (in red) denotes that it is denied. The neural network also consists of two hidden layers with two nodes each (in gray).

Now, let us assume that this neural network is trained to deny requests for large credit amounts from older people. Otherwise, the network should not discriminate with respect to age for small credit amounts. There should also not be discrimination for younger people with respect to the requested credit amount. When choosing age as the sensitive input, our approach can certify fairness with respect to different age groups for small credit amounts. Our approach is also able to find (as well as quantify) bias with respect to age for large credit amounts. Note that, in general, bias found by the analysis may be intended or accidental — our analysis does not aim to address this question. It is up to the user to decide whether the result of the analysis is expected or not.

Our approach does not require age to be an input of the neural network. For example, $x_{0,2}$ could denote the ZIP code of credit applicants, and the network could still use it as proxy for age. That is, requests for large credit amounts are denied for a certain range of ZIP codes (where older people tend to live), yet there is no discrimination between ZIP codes for small credit amounts. When choosing the ZIP code as the sensitive input, our approach would similarly be able to detect bias with respect to it for large credit amounts. In general, the choice of the sensitive features is up to the user, e.g., it can be driven by knowledge of proxies coming from a separate correlation analysis.

Below, we present on a high level how our approach achieves these results.

Naïve Approach. In theory, the simplest way to certify dependency fairness is to first analyze the neural network backwards starting from each output node, in our case $x_{3,1}$ and $x_{3,2}$. This allows us to determine the regions of the input space (i.e., age and requested credit amount) for which credit is approved and denied. For example, assume that we find that requests are denied for credit amounts larger than $10\,000$ (i.e., $10\,000 < x_{0,1}$) and age greater than 60 (i.e., $60 < x_{0,2}$), while they are approved for $x_{0,1} \leq 10\,000$ and $60 < x_{0,2}$ or for $x_{0,2} \leq 60$.

The second step is to forget the value of the sensitive input (i.e., age) or, in other words, to project these regions over the credit amount. In our example, after projection we have that credit requests are denied for $10\,000 < x_{0,1}$ and approved for any value of $x_{0,1}$. A non-empty intersection between the projected input regions indicates bias with respect to the sensitive input. In our example, the intersection is non-empty for $10\,000 < x_{0,1}$: there exist people that differ in age but request the same credit amount (greater than $10\,000$), some of whom receive the credit while others do not.

This approach, however, is not practical. Specifically, in neural networks with ReLU activation functions (see Section 3 for more details, other activation functions are discussed in Section 9), each hidden node effectively represents a disjunction between two activation statuses (active and inactive). In our example, there are 2^4 possible activation patterns for the 4 hidden nodes. To retain maximum precision, the analysis would have to explore all of them, which does not scale in practice.

Our Approach. Our analysis is based on the observation that *there might exist many activation patterns that do not correspond to a region of the input space* [Hanin and Rolnick 2019]. Such patterns can, therefore, be ignored during the analysis. We push this idea further by defining *abstract activation patterns*, which fix the activation status of only certain nodes and thus represent sets of (concrete) activation patterns. Typically, *a relatively small number of abstract activation patterns is sufficient for covering the entire input space*, without necessarily representing and exploring all possible concrete patterns.

Identifying those patterns that definitely correspond to a region of the input space is only possible with a forward analysis. Hence, we combine a forward pre-analysis with a backward analysis. The pre-analysis partitions the input space into independent partitions corresponding to abstract activation patterns. Then, the backward analysis tries to prove fairness of the neural network for each such partition.

More specifically, we set an upper bound U on the number of tolerated disjunctions (i.e., on the number of nodes with an unknown activation status) per abstract activation pattern. Our forward pre-analysis uses a cheap abstract domain (e.g., the boxes domain [Cousot and Cousot 1976]) to *iteratively* partition the input space along the *non-sensitive* input dimensions to obtain *fair* input partitions (i.e., boxes). Each partition satisfies one of the following conditions: (a) its classification is already fair because only one network output is reachable for all inputs in the region, (b) it has an abstract activation pattern with at most U unknown nodes, or (c) it needs to be partitioned further. We call partitions that satisfy condition (b) *feasible*.

In our example, let $\mathrm{U} = 2$. At first, the analysis considers the entire input space, that is, $\mathrm{x}_{0,1} : [0, 1]$ (credit amount) and $\mathrm{x}_{0,2} : [0, 1]$ (age). (Note that we could also specify a subset of the input space for analysis.) The abstract activation pattern corresponding to this initial partition I is ϵ (i.e., no hidden nodes have fixed activation status) and, thus, the number of disjunctions would be 4, which is greater than U. Therefore, I needs to be divided into I_1 ($\mathrm{x}_{0,1} : [0, 0.5].\mathrm{x}_{0,2} : [0, 1]$) and I_2 ($\mathrm{x}_{0,1} : [0.5, 1].\mathrm{x}_{0,2} : [0, 1]$). Observe that the input space is not split with respect to $\mathrm{x}_{0,2}$, which is the sensitive input. Now, I_1 is feasible since its abstract activation pattern is $\mathrm{x}_{1,2}\mathrm{x}_{2,1}\mathrm{x}_{2,2}$ (i.e., 3 nodes are always active), while I_2 must be divided further since its abstract activation pattern is ϵ.

To control the number of partitions, we impose a lower bound L on the size of each of their dimensions. Partitions that require a dimension of a smaller size are *excluded*. In other words, they are not considered until more analysis *budget* becomes available, that is, a larger U or a smaller L.

In our example, let $\mathrm{L} = 0.25$. The forward pre-analysis further divides I_2 into $\mathrm{I}_{2,1}$ ($\mathrm{x}_{0,1} : [0.5, 0.75].\mathrm{x}_{0,2} : [0, 1]$) and $\mathrm{I}_{2,2}$ ($\mathrm{x}_{0,1} : [0.75, 1].\mathrm{x}_{0,2} : [0, 1]$). Now, $\mathrm{I}_{2,1}$ is feasible, with abstract pattern $\mathrm{x}_{1,2}\mathrm{x}_{2,1}$, while $\mathrm{I}_{2,2}$ is not. However, $\mathrm{I}_{2,2}$ may not be split further because the size of the only non-sensitive dimension $\mathrm{x}_{0,1}$ has already reached the lower bound L. As a result, $\mathrm{I}_{2,2}$ is excluded, and only the remaining 75% of the input space is considered for analysis.

Next, feasible input partitions (within bounds L and U) are grouped by abstract activation patterns. In our example, the pattern corresponding to I_1, namely $\mathrm{x}_{1,2}\mathrm{x}_{2,1}\mathrm{x}_{2,2}$, is subsumed by the (more abstract) pattern of $\mathrm{I}_{2,1}$, namely $\mathrm{x}_{1,2}\mathrm{x}_{2,1}$. Consequently, we group I_1 and $\mathrm{I}_{2,1}$ under pattern $\mathrm{x}_{1,2}\mathrm{x}_{2,1}$.

The backward analysis is then run *in parallel* for each representative abstract activation pattern, in our example $\mathrm{x}_{1,2}\mathrm{x}_{2,1}$. This analysis determines the region of the input space (within a given

partition group) for which each output of the neural network is returned, e.g., credit is approved for $c_1 \leq \mathrm{x}_{0,1} \leq c_2$ and $a_1 \leq \mathrm{x}_{0,2} \leq a_2$. To achieve this, the analysis uses an expensive abstract domain, for instance, disjunctive or powerset polyhedra [Cousot and Cousot 1979; Cousot and Halbwachs 1978], and leverages abstract activation patterns to avoid disjunctions. For instance, pattern $\mathrm{x}_{1,2}\mathrm{x}_{2,1}$ only requires reasoning about two disjunctions from the remaining hidden nodes $\mathrm{x}_{1,1}$ and $\mathrm{x}_{2,2}$.

Finally, fairness is checked for each partition in the same way that it is done by the naïve approach for the entire input space. In our example, we prove that the classification within I_1 is fair and determine that within $\mathrm{I}_{2,1}$ the classification is biased. Concretely, our approach determines that bias occurs for $0.54 \leq \mathrm{x}_{0,1} \leq 0.75$, which corresponds to 21% of the entire input space (assuming a uniform probability distribution). In other words, the network returns different outputs for people that request the same credit in the above range but differ in age. Recall that partition $\mathrm{I}_{2,2}$, where $0.75 \leq \mathrm{x}_{0,1} \leq 1$, was excluded from analysis, and therefore, we cannot draw any conclusions about whether there is any bias for people requesting credit in this range.

Note that bias may also be quantified according to a probability distribution of the input space. In particular, it might be that credit requests in the range $0.54 \leq \mathrm{x}_{0,1} \leq 0.75$ are more (resp. less) common in practice. Given their probability distribution, our analysis computes a tailored percentage of bias, which in this case would be greater (resp. less) than 21%.

3 FEED-FORWARD DEEP NEURAL NETWORKS

Formally, a *feed-forward deep neural network* consists of an input layer (L_0), an output layer (L_N), and a number of hidden layers ($\mathrm{L}_1, \ldots, \mathrm{L}_{\mathrm{N}-1}$) in between. Each layer L_i contains $|\mathrm{L}_i|$ nodes and, with the exception of the input layer, is associated to a $|\mathrm{L}_i| \times |\mathrm{L}_{i-1}|$-matrix W_i of weight coefficients and a vector B_i of $|\mathrm{L}_i|$ bias coefficients. In the following, we use X to denote the set of all nodes, X_i to denote the set of nodes of the ith layer, and $\mathrm{x}_{i,j}$ to denote the jth node of the ith layer of a neural network. We focus here on neural networks used for *classification* tasks. Thus, $|\mathrm{L}_\mathrm{N}|$ is the number of target classes (e.g., 2 classes in Figure 1).

The value of the input nodes is given by the input data: continuous data is represented by one input node (e.g., $x_{0,1}$ or $x_{0,2}$ in Figure 1), while categorical data is represented by multiple input nodes via one-hot encoding. In the following, we use K to denote the subset of input nodes considered (directly or indirectly) *sensitive* to bias (e.g., $x_{0,2}$ in Figure 1) and $\overline{\mathrm{K}} \stackrel{\text{def}}{=} \mathrm{X}_0 \setminus \mathrm{K}$ to denote the input nodes not deemed sensitive to bias.

The value of each hidden and output node $\mathrm{x}_{i,j}$ is computed by an *activation function* f applied to a linear combination of the values of all nodes in the preceding layer [Goodfellow et al. 2016], i.e., $\mathrm{x}_{i,j} = f\left(\sum_k^{|\mathrm{L}_{i-1}|} \mathrm{w}^i_{j,k} \cdot \mathrm{x}_{i-1,k} + \mathrm{b}_{i,j}\right)$, where $\mathrm{w}^i_{j,k}$ and $\mathrm{b}_{i,j}$ are weight and bias coefficients in W_i and B_i, respectively. In a *fully-connected neural network*, all $\mathrm{w}^i_{j,k}$ are non-zero. Weights and biases are adjusted during the *training phase* of the neural network. In what follows, we focus on already trained neural networks, which we call *neural-network models*.

Nowadays, the most commonly used activation for hidden nodes is the Rectified Linear Unit (ReLU) [Nair and Hinton 2010]: $\mathrm{ReLU}(x) = \max(x, 0)$. In this case, the activation used for output nodes is the identity function. The output values are then normalized into a probability distribution on the target classes [Goodfellow et al. 2016]. We discuss other activation functions in Section 9.

4 TRACE SEMANTICS

In the following, we represent neural-network models as sequential programs. These programs consist of assignments for computing the activation value of each node (e.g., $x_{1,1} = -0.31 * x_{0,1} + 0.99 * x_{0,2} - 0.63$ in Figure 1) and implementations of activation functions (e.g., if-statements for

ReLUs). As is standard practice in static program analysis, we define a semantics for these programs that is tailored to our property of interest, dependency fairness, and use it to prove soundness of our approach. In particular, this allows us to formally prove soundness of the *parallelization* of our analysis as well (cf. Section 9), unlike for most of the existing approaches in which the parallelization is only introduced as part of the implementation and never proven sound.

The *semantics* of a neural-network model is a mathematical characterization of its behavior when executed for all possible input data. We model the operational semantics of a feed-forward neural-network model M as a transition system $\langle \Sigma, \tau \rangle$, where Σ is a (finite but exceedingly large) set of states and the *acyclic* transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the possible transitions between states [Cousot 2002; Cousot and Cousot 1977].

More specifically, a state $s \in \Sigma$ maps neural-network nodes to their values. Here, for simplicity, we assume that nodes have real values, i.e., $s : \mathrm{X} \to \mathbb{R}$. (We discuss floating-point values in Section 9.) In the following, we often only care about the values of a subset of the neural-network nodes in certain states. Thus, let $\Sigma_{|_Y} \stackrel{\text{def}}{=} \left\{ s_{|_Y} \mid s \in \Sigma \right\}$ be the restriction of Σ to a domain of interest Y. Sets $\Sigma_{|_{\mathrm{X}_0}}$ and $\Sigma_{|_{\mathrm{X}_\mathrm{N}}}$ denote restrictions of Σ to the network nodes in the input and output layer, respectively. With a slight abuse of notation, let $\mathrm{X}_{i,j}$ denote $\Sigma_{|_{\{x_{i,j}\}}}$, i.e., the restriction of Σ to the singleton set containing $\mathrm{x}_{i,j}$. Transitions happen between states with different values for consecutive nodes in the same layer, i.e., $\tau \subseteq \mathrm{X}_{i,j} \times \mathrm{X}_{i,j+1}$, or between states with different values for the last and first node of consecutive layers of the network, i.e., $\tau \subseteq \mathrm{X}_{i,|\mathrm{L}_i|} \times \mathrm{X}_{i+1,0}$. The set $\Omega \stackrel{\text{def}}{=} \{ s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \notin \tau \}$ is the set of final states of the neural network. These are partitioned in a set of outcomes $\mathbb{O} \stackrel{\text{def}}{=} \left\{ \left\{ s \in \Omega \mid \max \mathrm{X}_\mathrm{N} = \mathrm{x}_{\mathrm{N},i} \right\} \mid 0 \le i \le |\mathrm{L}_\mathrm{N}| \right\}$, depending on the output node with the highest value (i.e., the target class with highest probability).

Let $\Sigma^n \stackrel{\text{def}}{=} \{ s_0 \cdots s_{n-1} \mid \forall i < n : s_i \in \Sigma \}$ be the set of all sequences of exactly n states in Σ. Let $\Sigma^+ \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}^+} \Sigma^n$ be the set of all non-empty finite sequences of states. A *trace* is a sequence of states that respects the transition relation τ, i.e., $\langle s, s' \rangle \in \tau$ for each pair of consecutive states s, s' in the sequence. We write $\overline{\Sigma}^n$ for the set of all traces of n states: $\overline{\Sigma}^n \stackrel{\text{def}}{=} \{ s_0 \cdots s_{n-1} \in \Sigma^n \mid \forall i < n-1 : \langle s_i, s_{i+1} \rangle \in \tau \}$. The *trace semantics* $\Upsilon \in \mathcal{P}(\Sigma^+)$ generated by a transition system $\langle \Sigma, \tau \rangle$ is the set of all non-empty traces terminating in Ω [Cousot 2002]:

$$\Upsilon \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}^+} \left\{ s_0 \dots s_{n-1} \in \overline{\Sigma}^n \mid s_{n-1} \in \Omega \right\} \tag{1}$$

In the rest of the paper, we write $[\![\mathrm{M}]\!]$ to denote the trace semantics of a neural-network model M.

The trace semantics fully describes the behavior of M. However, reasoning about a certain property of M does not need all this information and, in fact, is facilitated by the design of a semantics that abstracts away from irrelevant details. In the following sections, we formally define our property of interest, dependency fairness, and systematically derive, using *abstract interpretation* [Cousot and Cousot 1977], a semantics tailored to reasoning about this property.

5 DEPENDENCY FAIRNESS

A *property* is specified by its extension, that is, by the set of elements having such a property [Cousot and Cousot 1977, 1979]. Properties of neural-network models are properties of their semantics. Thus, properties of network models with trace semantics in $\mathcal{P}(\Sigma^+)$ are sets of sets of traces in $\mathcal{P}(\mathcal{P}(\Sigma^+))$. In particular, the set of neural-network properties forms a complete boolean lattice $\langle \mathcal{P}(\mathcal{P}(\Sigma^+)), \subseteq, \cup, \cap, \emptyset, \mathcal{P}(\Sigma^+) \rangle$ for subset inclusion, that is, logical implication. The strongest property is the standard *collecting semantics* $\Lambda \in \mathcal{P}(\mathcal{P}(\Sigma^+))$:

$$\Lambda \stackrel{\text{def}}{=} \{ \Upsilon \} \tag{2}$$

Let $(\!|\mathrm{M}|\!)$ denote the collecting semantics of a particular neural-network model M. Then, model M satisfies a given property $\mathcal{H}$ if and only if its collecting semantics is a subset of $\mathcal{H}$:

$$\mathrm{M} \models \mathcal{H} \Leftrightarrow (\!|\mathrm{M}|\!) \subseteq \mathcal{H} \tag{3}$$

Here, we consider the property of *dependency fairness*, which expresses that the classification determined by a network model does not depend on sensitive input data. In particular, the property might interest the classification of all or just a fraction of the input space.

More formally, let $\mathbb{V}$ be the set of all possible value choices for all sensitive input nodes in K, e.g., for $\mathrm{K} = \{\mathrm{x}_{0,i}, \mathrm{x}_{0,j}\}$ one-hot encoding, say, gender information, $\mathbb{V} = \{\{1,0\},\{0,1\}\}$; for $\mathrm{K} = \{x_{0,k}\}$ encoding continuous data, say, in the range $[0,1]$, we use binning so a possibility is, e.g., $\mathbb{V} = \{[0,0.25],[0.25,0.75],[0.75,1]\}$. In the following, given a trace $\sigma \in \mathcal{P}(\Sigma^+)$, we write σ_0 and σ_ω to denote its initial and final state, respectively. We also write $\sigma_0 =_{\overline{\mathrm{K}}} \sigma'_0$ to indicate that the states σ_0 and σ'_0 agree on all values of all non-sensitive input nodes, and $\sigma_\omega \equiv \sigma'_\omega$ to indicate that σ and σ' have the same outcome $\mathrm{O} \in \mathbb{O}$. We can now formally define when the sensitive input nodes in K are *unused* with respect to a set of traces $T \in \mathcal{P}(\Sigma^+)$ [Urban and Müller 2018]. For one-hot encoded sensitive inputs[2], we have

$$\textsc{unused}_{\mathrm{K}}(T) \stackrel{\text{def}}{=} \forall \sigma \in T, \mathrm{V} \in \mathbb{V} \colon \sigma_0(\mathrm{K}) \neq \mathrm{V} \Rightarrow \exists \sigma' \in T \colon \sigma_0 =_{\overline{\mathrm{K}}} \sigma'_0 \wedge \sigma'_0(\mathrm{K}) = \mathrm{V} \wedge \sigma_\omega \equiv \sigma'_\omega, \tag{4}$$

where $\sigma_0(\mathrm{K}) \stackrel{\text{def}}{=} \{\sigma_0(\mathrm{x}) \mid \mathrm{x} \in \mathrm{K}\}$ is the image of K under σ_0. Intuitively, the sensitive input nodes in K are unused if any possible outcome in T (i.e., any outcome σ_ω of any trace σ in T) is possible from all possible value choices for K (i.e., there exists a trace σ' in T for each value choice for K with the same outcome as σ). That is, each outcome is independent of the value choice for K.

Example 5.1. Let us consider again our example in Figure 1. We write $\langle c, a\rangle \rightsquigarrow o$ for a trace starting in a state with $\mathrm{x}_{0,1} = c$ and $\mathrm{x}_{0,2} = a$ and ending in a state where o is the node with the highest value (i.e., the output class). The sensitive input $\mathrm{x}_{0,2}$ (age) is *unused* in $T = \{\langle 0.5, a\rangle \rightsquigarrow \mathrm{x}_{3,2} \mid 0 \leq a \leq 1\}$. It is instead *used* in $T' = \{\langle 0.75, a\rangle \rightsquigarrow \mathrm{x}_{3,2} \mid 0 \leq a < 0.51\} \cup \{\langle 0.75, a\rangle \rightsquigarrow \mathrm{x}_{3,1} \mid 0.51 \leq a \leq 1\}$.

The dependency-fairness property $\mathcal{F}_{\mathrm{K}}$ can now be defined as $\mathcal{F}_{\mathrm{K}} \stackrel{\text{def}}{=} \{[\![M]\!] \mid \textsc{unused}_{\mathrm{K}}([\![M]\!])\}$, that is, as the set of all neural-network models (or rather, their semantics) that do not use the values of the sensitive input nodes for classification. In practice, the property might interest just a fraction of the input space, i.e., we define

$$\mathcal{F}_{\mathrm{K}}[Y] \stackrel{\text{def}}{=} \{[\![\mathrm{M}]\!]^Y \mid \textsc{unused}_{\mathrm{K}}([\![\mathrm{M}]\!]^Y)\}, \tag{5}$$

where $Y \in \mathcal{P}(\Sigma)$ is a set of initial states of interest and the restriction $T^Y \stackrel{\text{def}}{=} \{\sigma \in T \mid \sigma_0 \in Y\}$ only contains traces of $T \in \mathcal{P}(\Sigma^+)$ that start with a state in Y. Similarly, in the rest of the paper, we write $S^Y \stackrel{\text{def}}{=} \{T^Y \mid T \in S\}$ for the set of sets of traces restricted to initial states in Y. Thus, from Equation 3, we have the following:

Theorem 5.2. $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y] \Leftrightarrow (\!|\mathrm{M}|\!)^Y \subseteq \mathcal{F}_{\mathrm{K}}[Y]$

Proof. The proof follows trivially from Equation 3 and the definition of $\mathcal{F}_{\mathrm{K}}[Y]$ (cf. Equation 5) and $(\!|\mathrm{M}|\!)^Y$. □

6 DEPENDENCY SEMANTICS

We now use abstract interpretation to systematically derive, by successive abstractions of the collecting semantics Λ, a *sound and complete* semantics $\Lambda_{\rightsquigarrow}$ that contains only and exactly the information needed to reason about $\mathcal{F}_{\mathrm{K}}[Y]$.

[2]For continuous sensitive inputs, we can replace $\sigma_0(\mathrm{K}) \neq \mathrm{V}$ (resp. $\sigma'_0(\mathrm{K}) = \mathrm{V}$) with $\sigma_0(\mathrm{K}) \not\subseteq \mathrm{V}$ (resp. $\sigma'_0(\mathrm{K}) \subseteq \mathrm{V}$).

6.1 Outcome Semantics

Let $T_Z \stackrel{\text{def}}{=} \{\sigma \in T \mid \sigma_\omega \in Z\}$ be the set of traces of $T \in \mathcal{P}(\Sigma^+)$ that end with a state in $Z \in \mathcal{P}(\Sigma)$. As before, we write $S_Z \stackrel{\text{def}}{=} \{T_Z \mid T \in S\}$ for the set of sets of traces restricted to final states in Z. From the definition of $\mathcal{F}_\mathrm{K}[Y]$ (and in particular, from the definition of $\textsc{unused}_\mathrm{K}$, cf. Equation 4), we have:

Lemma 6.1. $(\!|\mathrm{M}|\!)^Y \subseteq \mathcal{F}_\mathrm{K}[Y] \Leftrightarrow \forall \mathrm{O} \in \mathbb{O}\colon (\!|\mathrm{M}|\!)^Y_\mathrm{O} \subseteq \mathcal{F}_\mathrm{K}[Y]$

Proof. Let $(\!|\mathrm{M}|\!)^Y \subseteq \mathcal{F}_\mathrm{K}[Y]$. From the definition of $(\!|\mathrm{M}|\!)^Y$ (cf. Equation 2), we have that $[\![\mathrm{M}]\!]^Y \in \mathcal{F}_\mathrm{K}[Y]$. Thus, from the definition of $\mathcal{F}_\mathrm{K}[Y]$ (cf. Equation 5), we have $\textsc{unused}_\mathrm{K}([\![\mathrm{M}]\!]^Y)$. Now, from the definition of $\textsc{unused}_\mathrm{K}$ (cf. Equation 4), we equivalently have $\forall \mathrm{O} \in \mathbb{O}\colon \textsc{unused}_\mathrm{K}([\![\mathrm{M}]\!]^Y_\mathrm{O})$. Thus, we can conclude that $\forall \mathrm{O} \in \mathbb{O}\colon (\!|\mathrm{M}|\!)^Y_\mathrm{O} \subseteq \mathcal{F}_\mathrm{K}[Y]$. □

In particular, this means that in order to determine whether a neural-network model M satisfies dependency fairness, we can independently verify, for each of its possible target classes $\mathrm{O} \in \mathbb{O}$, that the values of its sensitive input nodes are unused.

We use this insight to abstract the collecting semantics Λ by *partitioning*. More specifically, let $\bullet \stackrel{\text{def}}{=} \left\{\Sigma^+_\mathrm{O} \mid \mathrm{O} \in \mathbb{O}\right\}$ be a trace partition with respect to outcome. We have the following Galois connection

$$\langle \mathcal{P}\left(\mathcal{P}\left(\Sigma^+\right)\right), \subseteq\rangle \underset{\alpha_\bullet}{\overset{\gamma_\bullet}{\leftrightarrows}} \langle \mathcal{P}\left(\mathcal{P}\left(\Sigma^+\right)\right), \mathrel{\dot\subseteq}\rangle, \tag{6}$$

where $\alpha_\bullet(S) \stackrel{\text{def}}{=} \{T_\mathrm{O} \mid T \in S \wedge \mathrm{O} \in \mathbb{O}\}$. The order $\mathrel{\dot\subseteq}$ is the pointwise ordering between sets of traces with the same outcome, i.e., $A \mathrel{\dot\subseteq} B \stackrel{\text{def}}{=} \bigwedge_{\mathrm{O}\in\mathbb{O}} \dot{A}_\mathrm{O} \subseteq \dot{B}_\mathrm{O}$, where $\dot{S}_Z$ denotes the only non-empty set of traces in S_Z. We can now define the *outcome semantics* $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^+))$ by abstraction of Λ:

$$\Lambda_\bullet \stackrel{\text{def}}{=} \alpha_\bullet(\Lambda) = \{\Upsilon_\mathrm{O} \mid \mathrm{O} \in \mathbb{O}\} \tag{7}$$

In the rest of the paper, we write $(\!|\mathrm{M}|\!)_\bullet$ to denote the outcome semantics of a particular neural-network model M.

6.2 Dependency Semantics

We observe that, to reason about dependency fairness, we do not need to consider all intermediate computations between the initial and final states of a trace. Thus, we can further abstract the outcome semantics into a set of dependencies between initial states and outcomes of traces.

To this end, we define the following Galois connection[3]

$$\langle \mathcal{P}\left(\mathcal{P}\left(\Sigma^+\right)\right), \mathrel{\dot\subseteq}\rangle \underset{\alpha_{\rightsquigarrow}}{\overset{\gamma_{\rightsquigarrow}}{\leftrightarrows}} \langle \mathcal{P}\left(\mathcal{P}\left(\Sigma \times \Sigma\right)\right), \mathrel{\dot\subseteq}\rangle, \tag{8}$$

where $\alpha_{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \{\{\langle\sigma_0, \sigma_\omega\rangle \mid \sigma \in T\} \mid T \in S\}$ [Urban and Müller 2018] abstracts away all intermediate states of any trace. We finally derive the *dependency semantics* $\Lambda_{\rightsquigarrow} \in \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$:

$$\Lambda_{\rightsquigarrow} \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(\Lambda_\bullet) = \{\{\langle\sigma_0, \sigma_\omega\rangle \mid \sigma \in \Upsilon_\mathrm{O}\} \mid \mathrm{O} \in \mathbb{O}\} \tag{9}$$

In the following, let $(\!|\mathrm{M}|\!)_{\rightsquigarrow}$ denote the dependency semantics of a particular network model M.

Let $R^Y \stackrel{\text{def}}{=} \{\langle s, \_\rangle \in R \mid s \in Y\}$ restrict a set of pairs of states to pairs whose first element is in Y and, similarly, let $S^Y \stackrel{\text{def}}{=} \left\{R^Y \mid R \in S\right\}$ restrict a set of sets of pairs of states to first elements in Y. The next result shows that $\Lambda_{\rightsquigarrow}$ is sound and complete for proving dependency fairness:

Theorem 6.2. $\mathrm{M} \models \mathcal{F}_\mathrm{K}[Y] \Leftrightarrow (\!|\mathrm{M}|\!)^Y_{\rightsquigarrow} \mathrel{\dot\subseteq} \alpha_{\rightsquigarrow}(\alpha_\bullet(\mathcal{F}_\mathrm{K}[Y]))$

[3]Note that here and in the following, for convenience, we abuse notation and reuse the order symbol $\mathrel{\dot\subseteq}$ defined over sets of sets of traces, instead of its abstraction, defined over sets of sets of pairs of states.

Proof. Let $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y]$. From Theorem 5.2, we have that $(\!|\mathrm{M}|\!)^{Y} \subseteq \mathcal{F}_{\mathrm{K}}[Y]$. Thus, from the Galois connections in Equation 6 and 8, we have $\alpha_{\rightsquigarrow}(\alpha_{\bullet}((\!|\mathrm{M}|\!)^{Y})) \sqsubseteq \alpha_{\rightsquigarrow}(\alpha_{\bullet}(\mathcal{F}_{\mathrm{K}}[Y]))$. From the definition of $(\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow}$ (cf. Equation 9), we can then conclude that $(\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow} \sqsubseteq \alpha_{\rightsquigarrow}(\alpha_{\bullet}(\mathcal{F}_{\mathrm{K}}[Y]))$. □

Corollary 6.3. $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y] \Leftrightarrow (\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow} \subseteq \alpha_{\rightsquigarrow}(\mathcal{F}_{\mathrm{K}}[Y])$

Proof. The proof follows trivially from the definition of $\sqsubseteq$ (cf. Equation 6 and 8) and Lemma 6.1. □

Furthermore, we observe that partitioning with respect to outcome induces a partition of the space of values of the input nodes *used* for classification. For instance, partitioning T' in Example 5.1 induces a partition on the values of (the indeed used node) $\mathrm{x}_{0,2}$. Thus, we can equivalently verify whether $(\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow} \subseteq \alpha_{\rightsquigarrow}(\mathcal{F}_{\mathrm{K}}[Y])$ by checking if the dependency semantics $(\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow}$ induces a partition of $Y_{|\overline{\mathrm{K}}}$. Let $R_0 \stackrel{\text{def}}{=} \{s \mid \langle s, \_\rangle \in R\}$ (resp. $R_\omega \stackrel{\text{def}}{=} \{s \mid \langle \_, s\rangle \in R\}$) be the selection of the first (resp. last) element from each pair in a set of pairs of states. We formalize this observation below.

Lemma 6.4. $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y] \Leftrightarrow \forall A, B \in (\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow} \colon (A_\omega \neq B_\omega \Rightarrow A_{0|_{\overline{\mathrm{K}}}} \cap B_{0|_{\overline{\mathrm{K}}}} = \emptyset)$

Proof. Let $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y]$. From Corollary 6.3, we have that $(\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow} \subseteq \alpha_{\rightsquigarrow}(\mathcal{F}_{\mathrm{K}}[Y])$. Thus, from the definition of $(\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow}$ (cf. Equation 9), we have $\forall \mathrm{O} \in \mathbb{O} \colon \alpha_{\rightsquigarrow}([\![\mathrm{M}]\!]^{Y}_{\mathrm{O}}) \in \alpha_{\rightsquigarrow}(\mathcal{F}_{\mathrm{K}}[Y])$. In particular, from the definition of $\alpha_{\rightsquigarrow}$ and $\mathcal{F}_{\mathrm{K}}[Y]$ (cf. Equation 5), we have that $\textsc{unused}_{\mathrm{K}}([\![\mathrm{M}]\!]^{Y}_{\mathrm{O}})$ for each $\mathrm{O} \in \mathbb{O}$. From the definition of $\textsc{unused}_{\mathrm{K}}$ (cf. Equation 4), for each pair of *non-empty* $[\![\mathrm{M}]\!]^{Y}_{\mathrm{O}_1}$ and $[\![\mathrm{M}]\!]^{Y}_{\mathrm{O}_2}$ for different $\mathrm{O}_1, \mathrm{O}_2 \in \mathbb{O}$ (the case in which one or both are empty is trivial), it must necessarily be the value of the non-sensitive input nodes in $\overline{\mathrm{K}}$ that causes the different outcome O_1 or O_2. We can thus conclude that $\forall A, B \in (\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow} \colon (A_\omega \neq B_\omega \Rightarrow A_{0|_{\overline{\mathrm{K}}}} \cap B_{0|_{\overline{\mathrm{K}}}} = \emptyset)$. □

7 NAÏVE DEPENDENCY-FAIRNESS ANALYSIS

In this section, we present a first static analysis for dependency fairness that computes a *sound* over-approximation $\Lambda^{\natural}_{\rightsquigarrow}$ of the dependency semantics $\Lambda_{\rightsquigarrow}$, i.e., $\Lambda_{\rightsquigarrow} \sqsubseteq \Lambda^{\natural}_{\rightsquigarrow}$. This analysis corresponds to the naïve approach we discussed in Section 2. While it is too naïve to be practical, it is still useful for building upon later in the paper.

For simplicity, we consider ReLU activation functions. (We discuss extensions to other activation functions in Section 9.) The naïve static analysis is described in Algorithm 1. It takes as input (cf. Line 14) a neural-network model M, a set of sensitive input nodes K of M, a (representation of a) set of initial states of interest Y, and an abstract domain A to be used for the analysis. The analysis proceeds backwards for each outcome (i.e., each target class $\mathrm{x}_{\mathrm{N},j}$) of M (cf. Line 17) in order to determine an over-approximation of the initial states that satisfy Y and lead to $\mathrm{x}_{\mathrm{N},j}$ (cf. Line 18).

More specifically, the transfer function $\textsc{outcome}_{\mathrm{A}}[\![\mathrm{x}]\!]$ (cf. Line 2) modifies a given abstract-domain element to assume the given outcome x, that is, to assume that $\max \mathrm{X}_{\mathrm{N}} = \mathrm{x}$. The transfer functions $\overleftarrow{\textsc{relu}}_{\mathrm{A}}[\![\mathrm{x}_{i,j}]\!]$ and $\overleftarrow{\textsc{assign}}_{\mathrm{A}}[\![\mathrm{x}_{i,j}]\!]$ (cf. Line 5) respectively consider a ReLU operation and replace $\mathrm{x}_{i,j}$ with the corresponding linear combination of nodes in the preceding layer (see Section 3).

Finally, the analysis checks whether the computed over-approximations satisfy dependency fairness with respect to K (cf. Line 19). In particular, it checks whether they induce a partition of $Y_{|\overline{\mathrm{K}}}$ as observed for Lemma 6.4 (cf. Lines 7-13). If so, we have proved that M satisfies dependency fairness. If not, the analysis returns a set B of abstract-domain elements over-approximating the input regions in which bias might occur.

Theorem 7.1. If $\textsc{analyze}(\mathrm{M}, \mathrm{K}, Y, \mathrm{A})$ of Algorithm 1 returns $\textsc{true}, \emptyset$ then M satisfies $\mathcal{F}_{\mathrm{K}}[Y]$.

Algorithm 1 : A Naïve Backward Analysis

1: **function** BACKWARD(M, A, x)
2: $\quad a \leftarrow \text{OUTCOME}_A[\![x]\!](\text{NEW}_A)$
3: $\quad$ **for** $i \leftarrow n - 1$ **down to** 0 **do**
4: $\quad\quad$ **for** $j \leftarrow |L_i|$ **down to** 0 **do**
5: $\quad\quad\quad a \leftarrow \overleftarrow{\text{ASSIGN}}_A[\![x_{i,j}]\!](\overleftarrow{\text{RELU}}_A[\![x_{i,j}]\!]a)$
6: $\quad$ **return** a
7: **function** CHECK(O)
8: $\quad B \leftarrow \emptyset$ ▷ B: biased
9: $\quad$ **for all** $o_1, a_1 \in O$ **do**
10: $\quad\quad$ **for all** $o_2 \neq o_1, a_2 \in O$ **do**
11: $\quad\quad\quad$ **if** $a_1 \sqcap_{A_2} a_2 \neq \bot_{A_2}$ **then**
12: $\quad\quad\quad\quad B \leftarrow B \cup \{a_1 \sqcap_{A_2} a_2\}$
13: $\quad$ **return** B
14: **function** ANALYZE(M, K, Y, A)
15: $\quad O \leftarrow \dot{\emptyset}$
16: $\quad$ **for** $j \leftarrow 0$ **up to** $|L_N|$ **do** ▷ perfectly parallelizable
17: $\quad\quad a \leftarrow \text{BACKWARD}(M, A, x_{N,j})$
18: $\quad\quad O \leftarrow O \cup \left\{x_{N,j} \rightarrow (\text{ASSUME}_A[\![Y]\!]a)_{|\overline{K}}\right\}$
19: $\quad B \leftarrow \text{CHECK}(O)$
20: $\quad$ **return** $B = \emptyset, B$ ▷ fair: $B = \emptyset$, maybe biased: $B \neq \emptyset$

Proof (Sketch). ANALYZE(M, K, Y, A) in Algorithm 1 computes an *over-approximation a* of the regions of the input space that yield each target class $x_{N,j}$ (cf. Line 17). Thus, it actually computes an over-approximation $(\!|M|\!)^{Y\lightning}_{\rightsquigarrow}$ of the dependency semantics $(\!|M|\!)^{Y}_{\rightsquigarrow}$, i.e., $(\!|M|\!)^{Y}_{\rightsquigarrow} \Subset (\!|M|\!)^{Y\lightning}_{\rightsquigarrow}$. Thus, if $(\!|M|\!)^{Y\lightning}_{\rightsquigarrow}$ satisfies $\mathcal{F}_K[Y]$, i.e., $\forall A, B \in (\!|M|\!)^{Y\lightning}_{\rightsquigarrow} : (A_\omega \neq B_\omega \implies A_{0|\overline{K}} \cap B_{0|\overline{K}} = \emptyset)$ (according to Lemma 6.4, cf. Line 19), then by transitivity we can conclude that also $(\!|M|\!)^{Y\lightning}_{\rightsquigarrow}$ necessarily satisfies $\mathcal{F}_K[Y]$. □

In the analysis implementation, there is a tradeoff between performance and precision, which is reflected in the choice of abstract domain A and its transfer functions. Unfortunately, existing numerical abstract domains that are less expressive than polyhedra [Cousot and Halbwachs 1978] would make for a rather fast but too imprecise analysis. This is because they are not able to precisely handle constraints like $\max X_N = x$, which are introduced by $\text{OUTCOME}_A[\![x]\!]$ to partition with respect to outcome.

Furthermore, even polyhedra would not be precise enough in general. Indeed, each $\overleftarrow{\text{RELU}}_A[\![x_{i,j}]\!]$ would over-approximate what effectively is a conditional branch. Let $|M| \stackrel{\text{def}}{=} |L_1| + \cdots + |L_{N-1}|$ denote the number of hidden nodes (i.e., the number of ReLUs) in a model M. On the other side of the spectrum, one could use a disjunctive completion [Cousot and Cousot 1979] of polyhedra, thus keeping a separate polyhedron for each branch of a ReLU. This would yield a precise (in fact, exact) but extremely slow analysis: even with parallelization (cf. Line 16), each of the $|L_N|$ processes would have to effectively explore $2^{|M|}$ paths!

In the rest of the paper, we improve on this naïve analysis and show how far we can go all the while remaining exact by using disjunctive polyhedra.

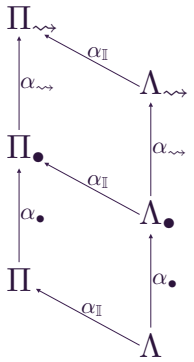


Fig. 2. Hierarchy of semantics.

8 PARALLEL SEMANTICS

We first have to take a step back and return to reasoning at the concrete-semantics level. At the end of Section 6, we observed that the dependency semantics of a neural-network model M satisfying $\mathcal{F}_{\mathrm{K}}[Y]$ effectively induces a partition of $Y_{|\overline{\mathrm{K}}}$. We call this input partition *fair*.

More formally, given a set Y of initial states of interest, we say that an input partition $\mathbb{I}$ of Y is fair if all value choices $\mathbb{V}$ for the sensitive input nodes K of M are possible in all elements of the partitions: $\forall \mathrm{I} \in \mathbb{I}, \mathrm{V} \in \mathbb{V} \colon \exists s \in \mathrm{I} \colon s(\mathrm{K}) = \mathrm{V}$. For instance, $\mathbb{I} = \{T, T'\}$, with T and T' in Example 5.1 is a fair input partition of $Y = \{s \mid s(\mathrm{x}_{0,1}) = 0.5 \vee s(\mathrm{x}_{0,1}) = 0.75\}$.

Given a fair input partition $\mathbb{I}$ of Y, the following result shows that we can verify whether a model M satisfies $\mathcal{F}_{\mathrm{K}}[Y]$ for each element I of $\mathbb{I}$, *independently*.

Lemma 8.1. $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y] \Leftrightarrow \forall \mathrm{I} \in \mathbb{I} \colon \forall A, B \in (\!|\mathrm{M}|\!)^{\mathrm{I}}_{\rightsquigarrow} \colon (A_\omega \neq B_\omega \Rightarrow A_{0|\overline{\mathrm{K}}} \cap B_{0|\overline{\mathrm{K}}} = \emptyset)$

Proof. The proof follows trivially from Lemma 6.4 and the fact that $\mathbb{I}$ is a fair partition. □

We use this new insight to further abstract the dependency semantics $\Lambda_{\rightsquigarrow}$. We have the following Galois connection

$$\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma)), \subseteq \rangle \underset{\alpha_{\mathbb{I}}}{\overset{\gamma_{\mathbb{I}}}{\leftrightarrows}} \langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma)), \subseteq_{\mathbb{I}} \rangle, \tag{10}$$

where $\alpha_{\mathbb{I}}(S) \overset{\text{def}}{=} \{R^{\mathrm{I}} \mid R \in S \wedge \mathrm{I} \in \mathbb{I}\}$. Here, the order $\subseteq_{\mathbb{I}}$ is the pointwise ordering between sets of pairs of states restricted to first elements in the same $\mathrm{I} \in \mathbb{I}$, i.e., $A \subseteq_{\mathbb{I}} B \overset{\text{def}}{=} \bigwedge_{\mathrm{I} \in \mathbb{I}} \dot{A}^{\mathrm{I}} \subseteq \dot{B}^{\mathrm{I}}$, where $\dot{S}^{\mathrm{I}}$ denotes the only non-empty set of pairs in S^{I}. We can now derive the *parallel semantics* $\Pi^{\mathbb{I}}_{\rightsquigarrow} \in \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$:

$$\Pi^{\mathbb{I}}_{\rightsquigarrow} \overset{\text{def}}{=} \alpha_{\mathbb{I}}(\Lambda_{\rightsquigarrow}) = \left\{ \left\{ \langle \sigma_0, \sigma_\omega \rangle \mid \sigma \in \Upsilon^{\mathrm{I}}_{\mathrm{O}} \right\} \mid \mathrm{I} \in \mathbb{I} \wedge \mathrm{O} \in \mathbb{O} \right\} \tag{11}$$

In fact, we derive a hierarchy of semantics, as depicted in Figure 2. We write $\{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow}$ to denote the parallel semantics of a particular neural-network model M. It remains to show soundness and completeness for $\Pi^{\mathbb{I}}_{\rightsquigarrow}$.

Theorem 8.2. $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y] \Leftrightarrow \{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow} \subseteq_{\mathbb{I}} \alpha_{\mathbb{I}}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}(\mathcal{F}_{\mathrm{K}}[Y])))$

Proof. Let $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y]$. From Theorem 6.2, we have that $(\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow} \subseteq \alpha_{\rightsquigarrow}(\alpha_{\bullet}(\mathcal{F}_{\mathrm{K}}[Y]))$. Thus, from the Galois connection in Equation 10, we have $\alpha_{\mathbb{I}}((\!|\mathrm{M}|\!)^{Y}_{\rightsquigarrow}) \subseteq \alpha_{\mathbb{I}}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}(\mathcal{F}_{\mathrm{K}}[Y])))$. From the definition of $\{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow}$ (cf. Equation 11), we can then conclude that $\{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow} \subseteq_{\mathbb{I}} \alpha_{\mathbb{I}}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}(\mathcal{F}_{\mathrm{K}}[Y])))$. □

Corollary 8.3. $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y] \Leftrightarrow \{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow} \subseteq \alpha_{\mathbb{I}}(\alpha_{\rightsquigarrow}(\mathcal{F}_{\mathrm{K}}[Y]))$

Proof. The proof follows trivially from the definition of $\subseteq_{\mathbb{I}}$ (cf. Equation 6, 8, and 10) and Lemma 6.1 and 8.1. □

Finally, from Lemma 8.1, we have that we can equivalently verify whether $\{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow} \subseteq \alpha_{\mathbb{I}}(\alpha_{\rightsquigarrow}(\mathcal{F}_{\mathrm{K}}[Y]))$ by checking if the parallel semantics $\{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow}$ induces a partition of each $\mathrm{I}_{|\overline{\mathrm{K}}}$.

Lemma 8.4. $\mathrm{M} \models \mathcal{F}_{\mathrm{K}}[Y] \Leftrightarrow \forall \mathrm{I} \in \mathbb{I}\colon \forall A, B \in \{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow}\colon (A^{\mathrm{I}}_{\omega} \neq B^{\mathrm{I}}_{\omega} \Rightarrow A^{\mathrm{I}}_{0|_{\overline{\mathrm{K}}}} \cap B^{\mathrm{I}}_{0|_{\overline{\mathrm{K}}}} = \emptyset)$

Proof. The proof follows trivially from Lemma 8.1. □

9 PARALLEL DEPENDENCY-FAIRNESS ANALYSIS

In this section, we build on the parallel semantics to design our novel *perfectly parallel* static analysis for dependency fairness, which automatically finds a fair partition $\mathbb{I}$ and computes a sound over-approximation $\Pi^{\mathbb{I}\natural}_{\rightsquigarrow}$ of $\Pi^{\mathbb{I}}{}_{\rightsquigarrow}$, i.e., $\Pi^{\mathbb{I}}{}_{\rightsquigarrow} \subseteq_{\mathbb{I}} \Pi^{\mathbb{I}\natural}_{\rightsquigarrow}$.

ReLU Activation Functions. We again only consider ReLU activation functions for now and postpone the discussion of other activation functions to the end of the section. The analysis is described in Algorithm 2. It combines a forward pre-analysis (Lines 15-24) with a backward analysis (Lines 28-38). The forward pre-analysis uses an abstract domain A_1 and builds partition $\mathbb{I}$, while the backward analysis uses an abstract domain A_2 and performs the actual dependency-fairness analysis of a neural-network model M with respect to its sensitive input nodes K and a (representation of a) set of initial states Y (cf. Line 13).

More specifically, the forward pre-analysis bounds the number of paths that the backward analysis has to explore. Indeed, not all of the $2^{|\mathrm{M}|}$ paths of a model M are necessarily viable starting from its input space.

In the rest of this section, we represent each path by an *activation pattern*, which determines the activation status of every ReLU operation in M. More precisely, an activation pattern is a sequence of flags. Each flag $\mathrm{p}_{i,j}$ represents the activation status of the ReLU operation used to compute the value of hidden node $\mathrm{x}_{i,j}$. If $\mathrm{p}_{i,j}$ is $\mathrm{x}_{i,j}$, the ReLU is always active, otherwise the ReLU is always inactive and $\mathrm{p}_{i,j}$ is $\overline{\mathrm{x}_{i,j}}$.

An *abstract activation pattern* gives the activation status of only a subset of the ReLUs of M, and thus, represents a set of activation patterns. ReLUs whose corresponding flag does not appear in an abstract activation pattern have an unknown (i.e., not fixed) activation status. Typically, *only a relatively small number of abstract activation patterns is sufficient for covering the entire input space of a neural-network model.* The design of our analysis builds on this key observation.

We set an analysis *budget* by providing an upper bound U (cf. Line 13) on the number of tolerated ReLUs with an unknown activation status for each element I of $\mathbb{I}$, i.e., on the number of paths that are to be explored by the backward analysis in each I. The forward pre-analysis starts with the trivial partition $\mathbb{I} = \{Y\}$ (cf. Line 15). It proceeds forward for each element I in $\mathbb{I}$ (cf. Lines 17-18). The transfer function $\overrightarrow{\mathrm{RELU}}^{\mathrm{p}}_{\mathrm{A}}[\![\mathrm{x}_{i,j}]\!]$ considers a ReLU operation and additionally builds an abstract activation pattern p for I (cf. Line 5) starting from the empty pattern ϵ (cf. Line 2).

If I leads to a unique outcome (cf. Line 19), then dependency fairness is already proved for I, and there is no need for a backward analysis; I is added to the set of *completed* partitions (cf. Line 20). Instead, if abstract activation pattern p fixes the activation status of enough ReLUs (cf. Line 21), we say that the backward analysis for I is *feasible*. In this case, the pair of p and I is inserted into a map F from abstract activation patterns to feasible partitions (cf. Line 22). The insertion takes care of merging abstract activation patterns that are subsumed by other (more) abstract patterns. In other words, it groups partitions whose abstract activation patterns fix more ReLUs with partitions whose patterns fix fewer ReLUs, and therefore, represent a superset of (concrete) patterns.

Algorithm 2 : Our Analysis Based on Activation Patterns

1: **function** FORWARD(M, A, I)
2: $a, p \leftarrow \text{ASSUME}_A[\![I]\!](\text{NEW}_A), \epsilon$
3: **for** $i \leftarrow 1$ **up to** N **do**
4: **for** $j \leftarrow 0$ **up to** $|L_i|$ **do**
5: $a, p \leftarrow \overrightarrow{\text{RELU}}^p_A[\![x_{i,j}]\!](\overrightarrow{\text{ASSIGN}}_A[\![x_{i,j}]\!]a)$
6: **return** a, p
7: **function** BACKWARD(M, A, O, p)
8: $a \leftarrow \text{OUTCOME}_A[\![O]\!](\text{NEW}_A)$
9: **for** $i \leftarrow N - 1$ **down to** 0 **do**
10: **for** $j \leftarrow |L_i|$ **down to** 0 **do**
11: $a \leftarrow \overleftarrow{\text{ASSIGN}}_A[\![x_{i,j}]\!](\overleftarrow{\text{RELU}}^p_A[\![x_{i,j}]\!]a)$
12: **return** a
13: **function** ANALYZE(M, K, Y, A_1, A_2, L, U)
14: $F, E, C \leftarrow \dot{\emptyset}, \dot{\emptyset}, \emptyset$ ▷ F: feasible, E: excluded, C: completed
15: $\mathbb{I} \leftarrow \{Y\}$
16: **while** $\mathbb{I} \neq \emptyset$ **do** ▷ perfectly parallelizable
17: $I \leftarrow \mathbb{I}.\text{GET}()$
18: $a, p \leftarrow \text{FORWARD}(M, A_1, I)$
19: **if** UNIQUELY-CLASSIFIED(a) **then** ▷ I is already fair
20: $C \leftarrow C \cup \{I\}$
21: **else if** $|M| - |p| \leq U$ **then** ▷ I is feasible
22: $F \leftarrow F \uplus \{p \rightarrow I\}$
23: **else if** $|I| \leq L$ **then** ▷ I is excluded
24: $E \leftarrow E \uplus \{p \rightarrow I\}$
25: **else** ▷ I must be partitioned further
26: $\mathbb{I} \leftarrow \mathbb{I} \cup \text{PARTITION}_{\overline{K}}(I)$
27: $B \leftarrow \emptyset$ ▷ B: biased
28: **for all** $p, \mathbb{I} \in F$ **do** ▷ perfectly parallelizable
29: $O \leftarrow \dot{\emptyset}$
30: **for** $j \leftarrow 0$ **up to** $|L_N|$ **do**
31: $a \leftarrow \text{BACKWARD}(M, A_2, x_{N,j}, p)$
32: $O \leftarrow O \cup \{x_{N,j} \rightarrow a\}$
33: **for all** $I \in \mathbb{I}$ **do**
34: $O' \leftarrow \dot{\emptyset}$
35: **for all** $o, a \in O$ **do**
36: $O' \leftarrow O' \cup \{o \rightarrow (\text{ASSUME}_{A_2}[\![I]\!]a)|_{\overline{K}}\}$
37: $B \leftarrow B \cup \text{CHECK}(O')$
38: $C \leftarrow C \cup \{I\}$
39: **return** $C, B = \emptyset, B, E$ ▷ fair: $B = \emptyset$, maybe biased: $B \neq \emptyset$

Otherwise, I needs to be partitioned further, with respect to $\overline{K}$ (cf. Line 25). Partitioning may continue until the size of I is smaller than the given lower bound L (cf. Lines 13 and 23). At this

point, I is set aside and excluded from the analysis until more resources (a larger upper bound U or a smaller lower bound L) become available (cf. Line 24).

Note that the forward pre-analysis lends itself to choosing a relatively cheap abstract domain A_1 since it does not need to precisely handle polyhedral constraints (like $\max \mathrm{X}_{\mathrm{N}} = \mathrm{x}$, needed to partition with respect to outcome, cf. Section 7).

The analysis then proceeds backwards, independently for each abstract activation path p and associated group of partitions $\mathbb{I}$ (cf. Lines 28 and 31). The transfer function $\overleftarrow{\textsc{relu}}^{\mathrm{p}}_{\mathrm{A}}[\![\mathrm{x}_{i,j}]\!]$ uses p to choose which path(s) to explore at each ReLU operation, i.e., only the active (resp. inactive) path if $\mathrm{x}_{i,j}$ (resp. $\overline{\mathrm{x}_{i,j}}$) appears in p, or both if the activation status of the ReLU corresponding to hidden node $\mathrm{x}_{i,j}$ is unknown. The (as we have seen, necessarily) expensive backward analysis only needs to run for each abstract activation pattern in the feasible map F. This is also why it is advantageous to merge subsumed abstract activation paths as described above.

Finally, the analysis checks dependency fairness of each element I associated to p (cf. Line 37). The analysis returns the set of input-space regions C that have been completed and a set B of abstract-domain elements over-approximating the regions in which bias might occur (cf. Line 39). If B is empty, then the given model M satisfies dependency fairness with respect to K and Y over C.

Theorem 9.1. If function $\textsc{analyze}(\mathrm{M}, \mathrm{K}, Y, \mathrm{A}_1, \mathrm{A}_2, \mathrm{L}, \mathrm{U})$ in Algorithm 2 returns C, TRUE, $\emptyset$, then M satisfies $\mathcal{F}_{\mathrm{K}}[Y]$ over the input-space fraction C.

Proof (Sketch). $\textsc{analyze}(\mathrm{M}, \mathrm{K}, Y, \mathrm{A}_1, \mathrm{A}_2, \mathrm{L}, \mathrm{U})$ in Algorithm 2 first computes the abstract activation patterns that cover a fraction C of the input space in which the analysis is feasible (Lines 15-24). Then, it computes an *over-approximation a* of the regions of C that yield each target class $\mathrm{x}_{\mathrm{N},j}$ (cf. Line 31). Thus, it actually computes an over-approximation $\{\!|\mathrm{M}|\!\}^{\mathbb{I}^\natural}_{\rightsquigarrow}$ of the parallel semantics $\{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow}$, i.e., $\{\!|\mathrm{M}|\!\}^{\mathbb{I}}_{\rightsquigarrow} \subseteq \{\!|\mathrm{M}|\!\}^{\mathbb{I}^\natural}_{\rightsquigarrow}$. Thus, if $\{\!|\mathrm{M}|\!\}^{\mathbb{I}^\natural}_{\rightsquigarrow}$ satisfies $\mathcal{F}_{\mathrm{K}}[Y]$, i.e., $\forall \mathrm{I} \in \mathbb{I}\colon \forall A, B \in \{\!|\mathrm{M}|\!\}^{\mathbb{I}^\natural}_{\rightsquigarrow}\colon (A^{\mathrm{I}}_{\omega} \neq B^{\mathrm{I}}_{\omega} \implies A^{\mathrm{I}}_{0|_{\overline{\mathrm{K}}}} \cap B^{\mathrm{I}}_{0|_{\overline{\mathrm{K}}}} = \emptyset)$ (according to Lemma 8.4, cf. Lines 33-37), then by transitivity we can conclude that also $\{\!|\mathrm{M}|\!\}^{\mathbb{I}^\natural}_{\rightsquigarrow}$ necessarily satisfies $\mathcal{F}_{\mathrm{K}}[Y]$. □

Remark. Recall that we assumed neural-network nodes to have real values (cf. Section 4). Thus, Theorem 9.1 is true for all choices of classical numerical abstract domains [Cousot and Cousot 1976; Cousot and Halbwachs 1978; Ghorbal et al. 2009; Miné 2006b, etc.] for A_1 and A_2. If we were to consider floating-point values instead, the only sound choices would be floating-point abstract domains [Chen et al. 2008; Miné 2004; Singh et al. 2019].

Other Activation Functions. Let us discuss how activation functions other than ReLUs would be handled. The only difference in Algorithm 2 would be the transfer functions $\overrightarrow{\textsc{relu}}^{\mathrm{p}}_{\mathrm{A}}[\![\mathrm{x}_{i,j}]\!]$ (cf. Line 5) and $\overleftarrow{\textsc{relu}}^{\mathrm{p}}_{\mathrm{A}}[\![\mathrm{x}_{i,j}]\!]$ (cf. Line 11), which would have to be replaced with the transfer functions corresponding to the considered activation function.

Piecewise-linear activation functions, like Leaky ReLU$(x) = \max(x, k \cdot x)$ or Hard TanH$(x) = \max(-1, \min(x, 1))$, can be treated analogously to ReLUs. The case of Leaky ReLUs is trivial. For Hard TanHs, the patterns p used in Algorithm 2 will consist of flags $\mathrm{p}_{i,j}$ with three possible values, depending on whether the corresponding hidden node $\mathrm{x}_{i,j}$ has value less than or equal to -1, greater than or equal to 1, or between -1 and 1. For these activation functions, our approach remains sound and, in practice, exact when using disjunctive polyhedra for the backward analysis.

Other activation functions, e.g., Sigmoid$(x) = \frac{1}{1+e^{-x}}$, can be soundly over-approximated [Singh et al. 2019] and similarly treated in a piecewise manner. In this case, however, we necessarily lose the exactness of the analysis, even when using disjunctive polyhedra.

10 IMPLEMENTATION

We implemented our dependency-fairness analysis described in the previous section in a tool called LIBRA. The implementation is written in PYTHON and is open source[4].

Tool Inputs. LIBRA takes as input a neural-network model M expressed as a PYTHON program (cf. Section 3), a specification of the input layer L_0 of M, an abstract domain for the forward pre-analysis, and budget constraints L and U. The specification for L_0 determines which input nodes correspond to continuous and (one-hot encoded) categorical data and, among them, which should be considered bias sensitive. We assume that continuous data is in the range $[0, 1]$. A set Y of initial states of interest is specified using an assumption at the beginning of the program representation of M.

Abstract Domains. For the forward pre-analysis, choices of the abstract domain are either boxes [Cousot and Cousot 1976] (i.e., BOXES in the following), or a combination of boxes and symbolic constant propagation [Li et al. 2019; Miné 2006a] (i.e., SYMBOLIC in the following), or the DEEPPOLY domain [Singh et al. 2019], which is designed for proving local robustness of neural networks. As previously mentioned, we use disjunctive polyhedra for the backward analysis. All abstract domains are built on top of the APRON abstract-domain library [Jeannet and Miné 2009].

Parallelization. Both the forward and backward analyses are parallelized to run on multiple CPU cores. The pre-analysis uses a queue from which each process draws a fraction I of Y (cf. Line 17). Fractions that need to be partitioned further are split in half along one of the non-sensitive dimensions (in a round-robin fashion), and the resulting (sub)fractions are put back into the queue (cf. Line 26). Feasible Is (with their corresponding abstract activation pattern p) are put into another queue (cf. Line 22) for the backward analysis.

Tool Outputs. The analysis returns the fractions of Y that were analyzed and any (sub)regions of these where bias was found. It also reports the percentage of the input space that was analyzed and (an estimate of) the percentage that was found biased according to a given probability distribution of the input space (uniform by default). To obtain the latter, we simply use the size of a box wrapped around each biased region. More precise but also costlier solutions exist [Barvinok 1994].

In general, how much of Y can be analyzed depends on the analysis configuration (i.e., chosen abstract domain for the forward pre-analysis and budget constraints L and U). For the fractions of Y that remained excluded from the analysis one can always re-run the analysis with a more powerful configuration, i.e., by restricting Y to the excluded fraction and choosing a more precise abstract domain or a higher L or a lower U. We plan on automating this process as part of our future work.

11 EXPERIMENTAL EVALUATION

In this section, we evaluate our approach by focusing on the following research questions:

RQ1: Can our analysis detect seeded (i.e., injected) bias?
RQ2: Is our analysis able to answer specific bias queries?
RQ3: How does the model structure affect the scalability of the analysis?
RQ4: How does the analyzed input-space size affect the scalability of the analysis?
RQ5: How does the analysis budget affect the scalability-vs-precision tradeoff?
RQ6: Can our analysis effectively leverage multiple CPUs?

[4] https://github.com/caterinaurban/Libra

11.1 Data

For our evaluation, we used public datasets from the UCI Machine Learning Repository and ProPublica (see below for more details) to train several neural-network models. We primarily focused on datasets discussed in the literature [Mehrabi et al. 2019] or used by related techniques (e.g., [Albarghouthi et al. 2017a,b; Albarghouthi and Vinitsky 2019; Bastani et al. 2019; Datta et al. 2017; Galhotra et al. 2017; Tramèr et al. 2017; Udeshi et al. 2018]) and recent work on fairness targeting feed-forward neural networks (e.g., [Manisha and Gujar 2020; Yurochkin et al. 2020]).

We pre-processed these datasets both to make them fair with respect to a certain sensitive input feature as well as to seed bias. This way, we eliminate as many sources of uncontrolled bias as possible, i.e., bias originally present in the dataset as well as bias potentially introduced by the training process due to under-representation in the dataset (as discussed in the Introduction). We describe how we seeded bias in each particular dataset later on in this section.

Our methodology for making the data fair was common across datasets. Specifically, given an original dataset and a sensitive feature (say, race), we selected the largest population with a particular value for this feature (say, Caucasian) from the dataset (and discarded all others). We removed any duplicate or inconsistent entries from this population. We then duplicated the population for every other value of the sensitive feature (say, Asian and Hispanic). For example, assuming the largest population was 500 Caucasians, we created 500 Asians and 500 Hispanics, and any two of these populations differ only in the value of race. Consequently, the new dataset is fair because there do not exist two inputs k and k' that differ only in the value of the sensitive feature for which the classification outcomes are different. A similar methodology is used by Yurochkin et al. in recent related work [Yurochkin et al. 2020].

We define the *unfairness score* of a dataset as the percentage of inputs k in the dataset for which there exists another input k' that differs from k only in the value of the sensitive feature and the classification outcome. Our fair datasets have an unfairness score of 0%.

All datasets used in our experiments are open source as part of LIBRA.

11.2 Setup

Since neural-network training is non-deterministic, we typically train eight neural networks on each dataset, unless stated otherwise. The model sizes range from 2 hidden layers with 5 nodes each to 32 hidden layers with 40 nodes each. All models were trained with Keras, using the RMSprop optimizer with the default learning rate, and categorical crossentropy as the loss function. Each model was trained for 50 iterations. With these settings, we generally obtain lower accuracy values than those reported in the literature (e.g., [Manisha and Gujar 2020]). We believe that this is largely due to the fact that we modified the original datasets to make them fair or to seed bias and, more importantly, the fact that we did not invest into hyperparameter optimizations. However, we remark and demonstrate below that training neural networks with higher accuracy is not needed to show that our analysis works (i.e., is indeed able to certify fairness or detect bias) and to evaluate its performance. All models used in our experiments are open source as part of LIBRA. For each model, we assume a uniform distribution of the input space.

We performed all experiments on a 12-core Intel ® Xeon ® X5650 CPU @ 2.67GHz machine with 48GB of memory, running Debian GNU/Linux 9.6 (stretch).

11.3 Results

In the following, we present our experimental results for each of the above research questions.

RQ1: Detecting Seeded Bias. This research question focuses on detecting seeded bias by comparing the analysis results for models trained with fair versus biased data.

Table 1. Analysis of Models Trained on Fair and {Age, Credit > 1000}-Biased Data (German Credit Data)

CREDIT	BOXES				SYMBOLIC				DEEPPOLY				
	FAIR DATA		BIASED DATA		FAIR DATA		BIASED DATA		FAIR DATA		BIASED DATA		
	BIAS	TIME	BIAS	TIME	BIAS	TIME	BIAS	TIME	BIAS	TIME	BIAS	TIME	
≤ 1000	0.09%	47s	0.09%	2m 17s	0.09%	13s	0.09%	1m 10s	0.09%	**10s**	0.09%	**39s**	MIN
	0.19%	5m 46s	0.45%	13m 2s	0.19%	**1m 5s**	0.45%	2m 41s	0.19%	1m 12s	0.45%	**1m 46s**	MEDIAN
	0.33%	30m 59s	0.95%	1h 56m 57s	0.33%	**4m 8s**	0.95%	**13m 16s**	0.33%	5m 45s	0.95%	18m 18s	MAX
> 1000	2.21%	1m 42s	4.52%	21m 11s	2.21%	**38s**	4.52%	**3m 7s**	2.21%	39s	4.52%	4m 44s	MIN
	6.72%	31m 42s	23.41%	1h 36m 51s	6.72%	8m 59s	23.41%	41m 44s	6.63%	**4m 58s**	23.41%	**15m 39s**	MEDIAN
	14.96%	7h 7m 12s	33.19%	16h 50m 48s	14.96%	4h 16m 52s	33.19%	8h 5m 14s	14.96%	**1h 9m 45s**	31.17%	**6h 51m 50s**	MAX

For this experiment, we used the German Credit dataset[5]. This dataset classifies creditworthiness into two categories, "good" and "bad". An input feature is age, which we consider sensitive to bias. (Recall that this could also be an input feature that the user considers indirectly sensitive to bias.) We seeded bias in the fair dataset by randomly assigning a bad credit score to people of age 60 and above who request a credit amount of more than EUR 1 000 until we reached a 20% unfairness score of the dataset. The median classification accuracy of the models (17 inputs and 4 hidden layers with 5 nodes each) trained on fair and biased data was 71% and 65%, respectively.

To analyze these models, we set $L = 0$ to be sure to complete the analysis on 100% of the input space. The drawback with this is that the pre-analysis might end up splitting input partitions endlessly. To counteract, for each model, we chose the smallest upper bound U that did not cause this issue (i.e., we used values for U between 1 and 16). Table 1 shows the analysis results for the different choices of domain used for the forward pre-analysis. In particular, it shows whether the models are biased with respect to age for credit requests of 1 000 or less as well as for credit requests of over 1 000. Columns BIAS and TIME show the detected bias (in percentage of the entire input space) and the analysis running time. We show minimum, median, and maximum bias percentage and running time for each credit request group. For each line in Table 1, we highlighted the choice of the abstract domain that entailed the shortest analysis time.

In this case, we expect the analysis to detect the highest bias percentage for credit requests of over 1 000 and for the models trained on biased data. This is indeed what we obtain: the analysis finds $3.5x$ median bias for high credit amounts compared to models trained on fair data. This demonstrates that *our approach is able to effectively detect seeded bias.*

For models trained on fair data, we observe a maybe unexpected difference in the bias found for small credit amounts compared to large credit amounts. This is in part due to the fact that bias is given in percentage of the entire input space and not scaled with respect to the *analyzed* input space (small credit amounts correspond to a mere 4% of the input space). When scaling the bias percentage with respect to the analyzed input space, the difference is less marked: the median bias is 0.19% / 4% = 4.75% for small credit amounts and 6.72% / 96% = 7% (or 6.63% / 96% = 6.9% for the DEEPPOLY domain) for large credit amounts. The remaining difference indicates that the models contain bias that does not necessarily depend on the credit amount. The bias is introduced by the training process itself (as explained in the Introduction) and is not due to imprecision of our analysis. Recall that our approach is exact, and imprecision is only introduced when estimating the bias percentage (cf. Section 10). Similar considerations justify the difference in the bias found for small credit amounts for models trained on fair data compared to those trained on biased data.

Finally, for comparison, the analysis of models trained on the original dataset (with median accuracy of 74%) found 0.28% bias for small credit amounts and 17.7% bias for large credit amounts.

RQ2: Answering Bias Queries. To further evaluate the precision of our approach, we created queries concerning bias within specific groups of people, each corresponding to a subset of the

[5] https://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data)

Table 2. Queries on Models Trained on Fair and Race-Biased Data (ProPublica's COMPAS Data)

QUERY	BOXES				SYMBOLIC				DEEPPOLY				
	FAIR DATA		BIASED DATA		FAIR DATA		BIASED DATA		FAIR DATA		BIASED DATA		
	BIAS	TIME	BIAS	TIME	BIAS	TIME	BIAS	TIME	BIAS	TIME	BIAS	TIME	
AGE < 25 RACE BIAS?	0.22%	24m 32s	0.12%	14m 53s	0.22%	11m 34s	0.12%	**7m 14s**	0.22%	**5m 18s**	0.12%	8m 46s	MIN
	0.31%	1h 54m 48s	0.99%	57m 33s	0.32%	**36m 0s**	0.99%	20m 43s	0.32%	47m 16s	0.99%	**16m 38s**	MEDIAN
	2.46%	2h 44m 11s	8.33%	5h 29m 19s	2.46%	2h 17m 3s	8.50%	3h 34m 50s	2.12%	**1h 11m 43s**	6.48%	**2h 5m 5s**	MAX
MALE AGE BIAS?	2.60%	24m 14s	4.51%	34m 23s	2.64%	25m 13s	5.20%	29m 19s	2.70%	**19m 47s**	5.22%	**20m 51s**	MIN
	6.08%	1h 49m 42s	6.95%	2h 3m 39s	6.77%	**1h 1m 51s**	7.02%	1h 2m 26s	6.77%	1h 13m 31s	7.00%	**47m 28s**	MEDIAN
	8.00%	5h 56m 6s	12.56%	8h 26m 55s	8.40%	**2h 2m 22s**	12.71%	4h 55m 35s	8.84%	2h 20m 23s	12.88%	**3h 25m 21s**	MAX
CAUCASIAN PRIORS BIAS?	2.18%	2h 54m 18s	2.92%	46m 53s	2.18%	1h 20m 41s	2.92%	30m 23s	2.18%	**18m 26s**	2.92%	**15m 29s**	MIN
	2.95%	6h 56m 44s	4.21%	3h 50m 38s	2.95%	4h 12m 28s	4.21%	3h 32m 52s	2.95%	**2h 36m 1s**	4.21%	**1h 34m 7s**	MEDIAN
	5.36%	**45h 2m 12s**	6.98%	70h 50m 10s	5.36%	60h 53m 6s	6.98%	49h 51m 42s	5.36%	52h 10m 2s	6.95%	**17h 48m 22s**	MAX

entire input space. We used the COMPAS dataset[6] from ProPublica for this experiment. The data assigns a three-valued recidivism-risk score (high, medium, and low) indicating how likely criminals are to re-offend. The data includes both personal attributes (e.g., age and race) as well as criminal history (e.g., number of priors and violent crimes). As for RQ1, we trained models both on fair and biased data. Here, we considered race as the sensitive feature. We seeded bias in the fair data by randomly assigning high recidivism risk to African Americans until we reached a 20% unfairness score of the dataset. The median classification accuracy of the 3-class models (19 inputs and 4 hidden layers with 5 nodes each) trained on fair and biased data was 55% and 56%, respectively.

To analyze these models, we used a lower bound L of 0, and an upper bound U between 7 and 19. Table 2 shows the results of our analysis (i.e., columns shown as in Table 1) for three queries, each representing a different choice for the set K of sensitive features:

Q_A: Is there bias with respect to *race* for people younger than 25?
Q_B: Is there bias with respect to *age* for males?
Q_C: Is there bias with respect to the number of priors for Caucasians?

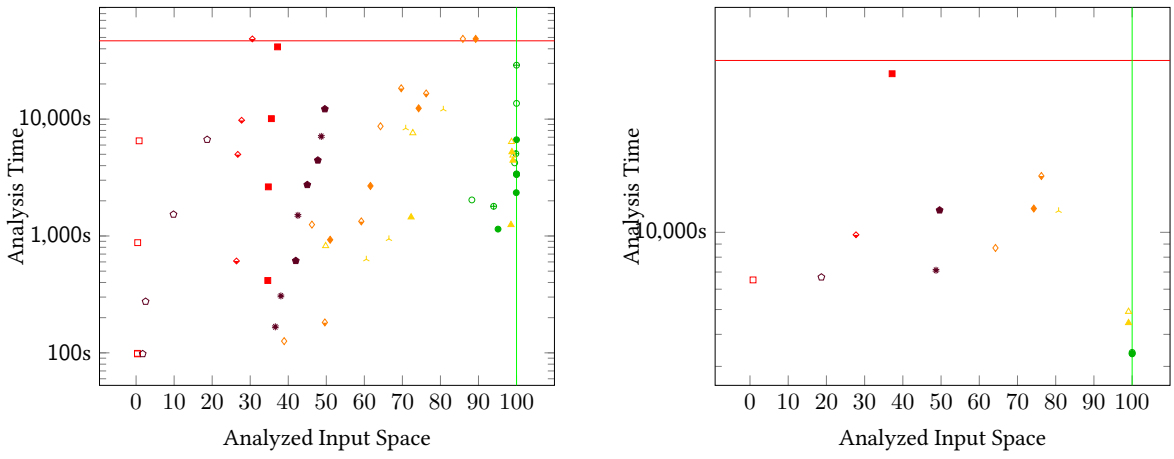
In this case, we expect the result of the analysis to change coherently according to the choice of K. Specifically, Q_A is expected to show a significant difference in the bias detected for models trained on fair data compared to models trained on biased data, while Q_B and Q_C should not show marked differences between fair and biased models. However, bias with respect to number of priors seems natural in the context of the dataset (i.e., recidivism risk increases with higher number of priors) and, therefore, we expect the bias percentage obtained for Q_C to be higher than that found for Q_B across both sets of fair and biased neural-network models.

The results in Table 2 meet our expectations. For Q_A, the analysis detects about three times as much median bias for the models trained on biased data compared to those trained on fair data. In contrast, for Q_B, the analysis finds a comparable amount of age bias across both sets of models. This becomes more evident when scaling the median bias with respect to the queried input space (males correspond to 50% of the input space): the smallest median bias for the models trained on fair data is 12.16% (for the BOXES domain) and the largest median bias for the models trained on biased data is 14.04% (for the SYMBOLIC domain). Finally, for Q_C, the analysis detects significant bias across both sets of models with respect to the number of priors. When considering the queried input space (Caucasians represent $1/6$ of the entire input space), this translates to 17.7% median bias for the models trained on fair data and 25.26% for the models trained on biased data. Overall, these results *demonstrate the effectiveness of our analysis in answering specific bias queries*. For comparison, the analysis of neural-network models trained on the original dataset (with median accuracy of 63%) found 1.21% median bias for Q_A, 5.34% median bias for Q_B, and 5.86% median bias for Q_C.

[6] https://www.propublica.org/datastore/dataset/compas-recidivism-risk-score-data-and-analysis

Table 3. Comparison of Different Model Structures (Adult Census Data)

|M|	U	BOXES					SYMBOLIC					DEEPPOLY				
		INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME
10 ○ ● ⊕	4	88.26%	1482	77	1136	33m 55s	95.14%	1132	65	686	19m 5s	93.99%	1894	77	992	29m 55s
	6	99.51%	769	51	723	1h 10m 25s	99.93%	578	47	447	39m 8s	99.83%	1620	54	1042	1h 24m 24s
	8	100.00%	152	19	143	3h 47m 23s	100.00%	174	18	146	1h 51m 2s	100.00%	1170	26	824	8h 2m 27s
	10	100.00%	1	1	1	**55m 58s**	100.00%	1	1	1	56m 8s	100.00%	1	1	1	56m 43s
12 △ ▲ ⅄	4	49.83%	719	9	329	13m 43s	72.29%	1177	11	559	24m 9s	60.52%	1498	14	423	10m 32s
	6	72.74%	1197	15	929	2h 6m 49s	98.54%	333	7	195	20m 46s	66.46%	1653	17	594	15m 44s
	8	98.68%	342	9	284	1h 46m 43s	98.78%	323	9	190	1h 27m 18s	70.87%	1764	18	724	2h 19m 11s
	10	99.06%	313	7	260	1h 21m 47s	99.06%	307	5	182	**1h 13m 55s**	80.76%	1639	18	1007	3h 22m 11s
20 ◇ ◆ ⬙	4	38.92%	1044	18	39	2m 6s	51.01%	933	31	92	15m 28s	49.62%	1081	34	79	3m 2s
	6	46.22%	1123	62	255	20m 51s	61.60%	916	67	405	44m 40s	59.20%	1335	90	356	22m 13s
	8	64.24%	1111	96	792	2h 24m 51s	74.27%	1125	78	780	3h 26m 20s	69.69%	1574	127	652	5h 6m 7s
	10	85.90%	1390	71	1339	>13h	89.27%	1435	60	1157	>13h	76.25%	1711	148	839	**4h 36m 23s**
40 □ ■ ⬙	4	0.35%	10	0	0	1m 39s	34.62%	768	1	1	6m 56s	26.39%	648	2	3	10m 11s
	6	0.35%	10	0	0	1m 38s	34.76%	817	4	5	43m 53s	26.74%	592	8	10	1h 23m 11s
	8	0.42%	12	1	2	14m 37s	35.56%	840	21	28	2h 48m 15s	27.74%	686	32	42	2h 43m 2s
	10	0.80%	23	10	13	1h 48m 43s	37.19%	880	50	75	**11h 32m 21s**	30.56%	699	83	121	>13h
45 ⬠ ⬟ ✳	4	1.74%	50	0	0	1m 38s	41.98%	891	14	49	10m 14s	36.60%	805	6	8	2m 47s
	6	2.50%	72	3	22	4m 35s	45.00%	822	32	143	45m 42s	38.06%	847	25	50	5m 7s
	8	9.83%	282	25	234	25m 30s	47.78%	651	46	229	1h 14m 5s	42.53%	975	74	180	25m 1s
	10	18.68%	522	33	488	1h 51m 24s	49.62%	714	51	294	**3h 23m 20s**	48.68%	1087	110	373	1h 58m 34s



(a) (b) Zoom on Best U-Configurations

Fig. 3. Comparison of Different Model Structures (Adult Census Data)

For each line in Table 2, we highlighted the choice of abstract domain that entailed the shortest analysis time. We observe that DEEPPOLY seems generally the better choice. The difference in performance becomes more striking as the analyzed input space becomes smaller, i.e., for Q_C. This is because DEEPPOLY is specifically designed for proving *local* robustness of neural networks. Thus, our input partitioning, in addition to allowing for parallelism, is also enabling analyses designed for local properties to prove global properties, like dependency fairness.

RQ3: Effect of Model Structure on Scalability. To evaluate the effect of the model structure on the scalability of our analysis, we trained models on the Adult Census dataset[7] by varying the number of layers and nodes per layer. The dataset assigns a yearly income (> or ≤ USD 50K) based on personal attributes such as gender, race, and occupation. All models (with 23 inputs) were trained on a fair dataset with respect to gender and had minimum classification accuracy of 78%.

Table 3 shows the results. The first column (|M|) shows the total number of hidden nodes and introduces the marker symbols used in the scatter plot of Figure 3 (to identify the domain used for

[7] https://archive.ics.uci.edu/ml/datasets/adult

the forward pre-analysis: left, center, and right symbols respectively refer to the BOXES, SYMBOLIC, and DEEPPOLY domains). The models have the following number of hidden layers and nodes per layer (from top to bottom): 2 and 5; 4 and 3; 4 and 5; 4 and 10; 9 and 5.

Column U shows the chosen upper bound for the analysis. For each model, we tried four different choices of U. Column INPUT shows the input-space coverage, i.e., the percentage of the input space that was completed by the analysis. Column |C| shows the total number of analyzed (i.e., completed) input space partitions. Column |F| shows the total number of abstract activation patterns (left) and feasible input partitions (right) that the backward analysis had to explore. The difference between |C| and the number of partitions shown in |F| are the input partitions that the pre-analysis found to be already fair (i.e., uniquely classified). Finally, column TIME shows the analysis running time. We used a lower bound L of 0.5 and a time limit of 13h. For each model in Table 3, we highlighted the configuration (i.e., domain used for the pre-analysis and chosen U) that achieved the highest input-space coverage (the analysis running time being decisive in case of equality or timeout).

The scatter plot of Figure 3a visualizes the input coverage and analysis running time. We zoom in on the best U-configurations for each pre-analysis domain (i.e., the chosen U) in Figure 3b.

Overall, we observe that *coverage decreases for larger model structures*, and the more precise SYMBOLIC and DEEPPOLY domains result in a significant coverage boost, especially for larger structures. We also note that, as in this case we are analyzing the entire input space, DEEPPOLY generally performs worse than the SYMBOLIC domain. In particular, for larger structures, *the SYMBOLIC domain often yields a higher input coverage in a shorter analysis running time*. Interestingly, the input coverage is higher for the largest model with 45 hidden nodes (i.e., 9 hidden layers with 5 nodes each) than that with 40 hidden nodes (i.e., 4 hidden layers with 10 nodes each). In fact, in neural-network models with more layers but fewer nodes per layer, the activation status of a node fixes the activation status of more nodes in subsequent layers. As a consequence, fewer activation patterns are actually needed to cover a larger portion of the input space. For instance with the SYMBOLIC domain only 14 patterns are enough to cover 41.98% of the input space for the model with 45 hidden nodes with U = 4, while for the model with 40 hidden nodes already 50 patterns are needed to cover only 37.19% of the input space with U = 10. Finally, we observe that *increasing the upper bound* U *tends to increase coverage independently of the specific model structure*. However, interestingly, this does not always come at the expense of an increased running time. In fact, such a change often results in decreasing the number of partitions that the expensive backward analysis needs to analyze (cf. columns |F|) and, in turn, this reduces the overall running time.

RQ4: Effect of Analyzed Input Space on Scalability. As said above, the analysis of the models considered in Table 3 is conducted *on the entire input space*. In practice, as already mentioned, one might be interested in just a portion of the input space, e.g., depending on the probability distribution. More generally, **we argue that the size of the analyzed input space (rather than the size of the analyzed neural network) is the most important factor that affects the performance of the analysis**. To support this claim, we trained even larger models and analyzed them with respect to queries exercising different input space sizes. Table 4 shows the results. The first column again shows the total number of hidden nodes for each trained model. In particular, the models we analyzed have the following number of hidden layers and nodes per layer (from top to bottom): 4 and 5; 8 and 10; 16 and 20; 32 and 40. Column QUERY shows the query used for the analysis and the corresponding queried input space size. Specifically, the queries identify people with the following characteristics:

A: TRUE queried input space: 100.0%

Table 4. Comparison of Different Input Space Sizes and Model Structures (Adult Census Data)

|M|	QUERY	BOXES					SYMBOLIC					DEEPPOLY				
		INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME
20	F	100.000% 0.009%	9	2	3	3m 3s	100.000% 0.009%	5	1	2	3m 5s	100.000% 0.009%	3	1	1	**2m 33s**
	E	99.996% 0.104%	83	9	39	3m 13s	100.000% 0.104%	26	3	9	3m 8s	100.000% 0.104%	22	3	9	**2m 38s**
	D	99.978% 1.042%	457	13	176	5m	100.000% 1.042%	292	9	63	**4m 50s**	100.000% 1.042%	287	6	65	5m 14s
	C	99.696% 8.333%	3173	20	1211	36m 12s	100.000% 8.333%	2668	13	417	**17m 40s**	100.000% 8.333%	2887	10	519	29m 52s
	B	97.318% 50%	15415	61	5646	1h 39m 36s	99.991% 49.996%	12617	34	2112	**1h 1m 19s**	99.978% 49.989%	13973	24	2405	1h 14m 19s
	A	94.032% 100%	18642	70	8700	2h 30m 46s	99.935% 99.935%	15445	40	3481	**1h 29m**	99.896% 99.896%	17784	39	4076	1h 47m 7s
80	F	99.931% 0.009%	11	0	0	3m 5s	99.961% 0.009%	17	0	0	**3m 2s**	99.957% 0.009%	10	0	0	2m 36s
	E	99.583% 0.104%	61	0	0	3m 6s	99.783% 0.104%	89	0	0	**3m 10s**	99.753% 0.104%	74	0	0	2m 44s
	D	97.917% 1.042%	151	0	0	2m 56s	99.258% 1.034%	297	0	0	**3m 41s**	98.984% 1.031%	477	0	0	2m 58s
	C	83.503% 8.333%	506	2	3	2h 1m	95.482% 7.956%	885	25	34	>13h	93.225% 7.768%	1145	23	33	**12h 57m 37s**
	B	25.634% 50%	5516	7	11	**1h 28m 6s**	76.563% 38.281%	4917	123	182	>13h	63.906% 31.953%	7139	117	152	>13h
	A	0.052% 100%	12	0	0	25m 51s	61.385% 61.385%	5156	73	102	**10h 25m 2s**	43.698% 43.698%	4757	68	88	>13h
320	F	99.931% 0.009%	6	0	0	3m 15s	99.944% 0.009%	9	0	0	**3m 35s**	99.931% 0.009%	6	0	0	3m 30s
	E	99.583% 0.104%	121	0	0	3m 39s	99.627% 0.104%	120	0	0	**6m 34s**	99.583% 0.104%	31	0	0	4m 22s
	D	97.917% 1.042%	151	0	0	6m 18s	98.247% 1.024%	597	0	0	**21m 9s**	97.917% 1.020%	301	0	0	9m 35s
	C	83.333% 8.333%	120	0	0	30m 37s	88.294% 7.358%	755	0	0	**1h 36m 35s**	83.342% 6.945%	483	0	0	52m 29s
	B	25.000% 50%	5744	0	0	2h 24m 36s	46.063% 23.032%	4676	0	0	**7h 25m 57s**	25.074% 12.537%	5762	4	4	>13h
	A	0.000% 100%	0	0	0	2h 54m 25s	24.258% 24.258%	2436	0	0	**9h 41m 36s**	0.017% 0.017%	4	0	0	5h 3m 33s
1280	F	99.931% 0.009%	11	0	0	7m 35s	99.948% 0.009%	10	0	0	**24m 42s**	99.931% 0.009%	6	0	0	7m 6s
	E	99.583% 0.104%	31	0	0	15m 49s	99.674% 0.104%	71	0	0	**51m 52s**	99.583% 0.104%	31	0	0	15m 14s
	D	97.917% 1.042%	151	0	0	1h 49s	98.668% 1.028%	557	0	0	**3h 31m 45s**	97.917% 1.020%	301	0	0	1h 3m 33s
	C	83.333% 6.944%	481	0	0	**7h 11m 39s**	–	–	–	–	>13h	83.333% 6.944%	481	0	0	7h 12m 57s
	B	–	–	–	–	>13h	–	–	–	–	>13h	–	–	–	–	>13h
	A	–	–	–	–	>13h	–	–	–	–	>13h	–	–	–	–	>13h

B: $A \wedge \text{age}^{8} \leq 53.5$ queried input space: 50.00%
C: $B \wedge$ race = white queried input space: 8.333% (3 race choices)
D: $C \wedge$ work class = private queried input space: 1.043% (4 work class choices)
E: $D \wedge$ marital status = single queried input space: 0.104% (5 marital status choices)
F: $E \wedge$ occupation = blue-collar queried input space: 0.009% (6 occupation choices)

For the analysis budget, we used L = 0.25, U = 0.1 ∗ |M|, and a time limit of 13h. Column INPUT shows, for each domain used for the forward pre-analysis, the coverage of the queried input space (i.e., the percentage of the input space that satisfies the query and was completed by the analysis) and the corresponding input-space coverage (i.e., the same percentage but this time scaled to the entire input space). Columns U, |C|, |F|, and TIME are as before. Where a timeout is indicated (i.e., TIME > 13h) and the values for the INPUT, |C|, and |F| columns are missing, it means that the timeout occurred during the pre-analysis; otherwise, it happened during the backward analysis. For each model and query, we highlighted the configuration (i.e., the abstract domain used for the

[8]This corresponds to $age \leq 0.5$ with min-max scaling between 0 and 1.

Table 5. Comparison of Different Analysis Configurations (Japanese Credit Screening) — 12 CPUs

L	U	⬟ BOXES					▲ SYMBOLIC					✶ DEEPPOLY				
		INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME
0.5	4	15.28%	37	0	0	8s	58.33%	79	8	20	1m 26s	69.79%	115	10	39	3m 18s
	6	17.01%	39	6	6	51s	69.10%	129	22	61	5m 41s	80.56%	104	23	51	7m 53s
	8	51.39%	90	28	85	12m 2s	82.64%	88	31	67	12m 35s	91.32%	84	27	56	19m 33s
	10	79.86%	89	34	89	34m 15s	93.06%	98	40	83	42m 32s	96.88%	83	29	58	43m 39s
0.25	4	59.09%	1115	20	415	54m 32s	95.94%	884	39	484	54m 31s	98.26%	540	65	293	14m 29s
	6	83.77%	1404	79	944	37m 19s	98.68%	634	66	376	23m 31s	99.70%	322	79	205	13m 25s
	8	96.07%	869	140	761	1h 7m 29s	99.72%	310	67	247	1h 3m 33s	99.98%	247	69	177	22m 52s
	10	99.54%	409	93	403	1h 35m 20s	99.98%	195	52	176	1h 2m 13s	100.00%	111	47	87	34m 56s
0.125	4	97.13%	12449	200	9519	3h 33m 48s	99.99%	1101	60	685	47m 46s	99.99%	768	81	415	19m 1s
	6	99.83%	5919	276	4460	3h 23m	100.00%	988	77	606	26m 47s	100.00%	489	80	298	16m 54s
	8	99.98%	1926	203	1568	2h 14m 25s	100.00%	404	73	309	46m 31s	100.00%	175	57	129	20m 11s
	10	100.00%	428	95	427	1h 39m 31s	100.00%	151	53	141	57m 32s	100.00%	80	39	62	28m 33s
0	4	100.00%	19299	295	15446	6h 13m 24s	100.00%	1397	60	885	40m 5s	100.00%	766	87	425	**16m 41s**
	6	100.00%	4843	280	3679	2h 24m 7s	100.00%	763	66	446	35m 24s	100.00%	401	81	242	32m 29s
	8	100.00%	1919	208	1567	2h 9m 59s	100.00%	404	73	309	45m 48s	100.00%	193	68	144	24m 16s
	10	100.00%	486	102	475	1h 41m 3s	100.00%	217	55	192	1h 2m 11s	100.00%	121	50	91	30m 53s

pre-analysis) that achieved the highest input-space coverage with the shortest analysis running time. Note that, where the |F| column only contains zeros, it means that the backward analysis had no activation patterns to explore; this implies that the entire covered input space (i.e., the percentage shown in the INPUT column) was already certified to be fair by the forward analysis.

Overall, we observe that *whenever the analyzed input space is small enough* (i.e., queries $D - F$), *the size of the neural network has little influence on the input space coverage* and slightly impacts the analysis running time, independently of the domain used for the forward pre-analysis. Instead, for larger analyzed input spaces (i.e., queries $A - C$) performance degrades quickly for larger neural networks. These results thus support our claim. In fact, these considerations generalize to other research areas in the verification of neural networks, e.g., in the certification of local robustness against adversarial examples: the size of the perturbation is the most important factor that affects the performance of the verification [Tran et al. 2020]. Finally, again, we observe that the SYMBOLIC domain generally is the better choice for the forward pre-analysis, in particular for queries exercising a larger input space or larger neural networks.

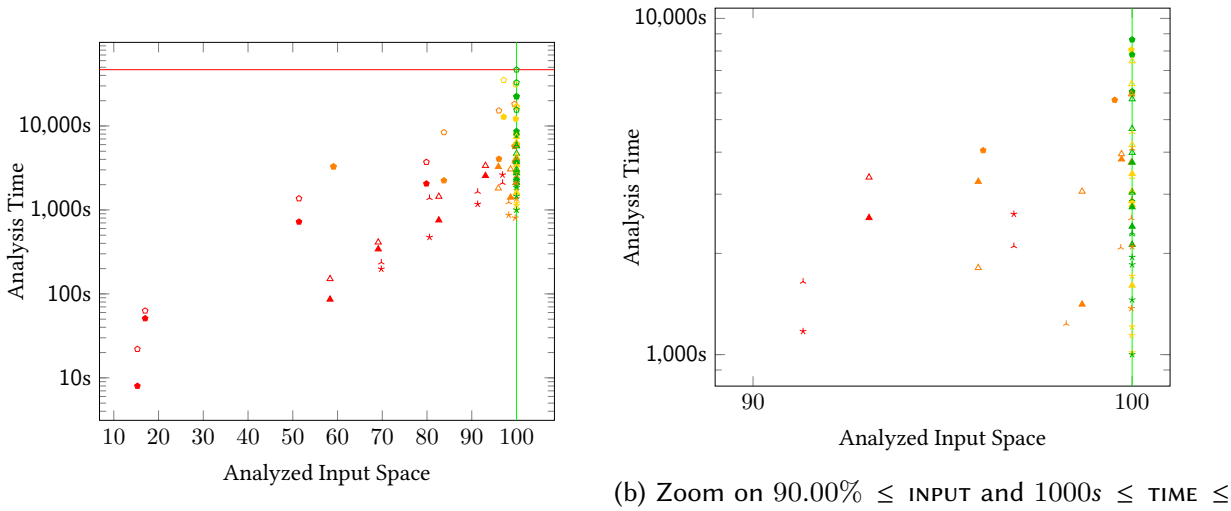
RQ5: Scalability-vs-Precision Tradeoff. To evaluate the effect of the analysis budget (bounds L and U), we analyzed a model using different budget configurations. For this experiment, we used the Japanese Credit Screening[9] dataset, which we made fair with respect to gender. Our 2-class model (17 inputs and 4 hidden layers with 5 nodes each) had a classification accuracy of 86%.

Table 5 shows the results of the analysis for different budget configurations and choices for the domain used for the forward pre-analysis. The best configuration in terms of input-space coverage and analysis running time is highlighted. The symbol next to each domain name introduces the marker used in the scatter plot of Figure 4a, which visualizes the coverage and running time. Figure 4b zooms on $90.00\% \leq$ INPUT and $1000s \leq$ TIME $\leq 1000s$.

Overall, we observe that *the more precise* SYMBOLIC *and* DEEPPOLY *domains boost input coverage*, most noticeably for configurations with a larger L. This additional precision does not always result in longer running times. In fact, a more precise pre-analysis often reduces the overall running time. This is because the pre-analysis is able to prove that more partitions are already fair without requiring them to go through the backward analysis (cf. columns |F|).

Independently of the chosen domain for the forward pre-analysis, as expected, *a larger* U *or a smaller* L *increase precision.* Increasing U or L typically reduces the number of completed partitions

[9] https://archive.ics.uci.edu/ml/datasets/Japanese+Credit+Screening



(a)

(b) Zoom on 90.00% ≤ INPUT and 1000*s* ≤ TIME ≤ 1000*s*

Fig. 4. Comparison of Different Analysis Configurations (Japanese Credit Screening)

Table 6. Comparison of Different Analysis Configurations (Japanese Credit Screening) — 4 CPUs

L	U	⬠ BOXES					△ SYMBOLIC					⅄ DEEPPOLY				
		INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME
0.5	4	15.28%	44	0	0	22s	58.33%	96	8	26		2m 31s	69.79%	85	9	30		3m 57s
	6	17.01%	40	5	5	1m 3s	69.10%	97	18	45		6m 52s	80.56%	131	26	63		23m 6s
	8	51.39%	96	29	88	22m 47s	82.64%	128	34	87		24m 5s	91.32%	117	34	78		27m 28s
	10	79.86%	109	36	107	1h 1m 54s	93.06%	104	36	92		56m 8s	96.88%	69	31	50		35m 2s
0.25	4	59.09%	1147	22	405	54m 51s	95.94%	715	43	407		30m 12s	98.26%	488	65	272		20m 35s
	6	83.77%	1757	80	1149	2h 19m 50s	98.68%	693	73	400		50m 57s	99.70%	322	79	205		34m 42s
	8	96.07%	1129	136	950	4h 13m 49s	99.72%	289	62	232		1h 5m 53s	99.98%	153	56	113		42m 25s
	10	99.54%	510	92	497	5h 3m 34s	99.98%	158	57	150		1h 39m 14s	100.00%	109	46	85		1h 8m 18s
0.125	4	97.13%	12398	200	9491	9h 46m	99.99%	1864	58	1188		1h 46m 25s	99.99%	1257	92	670		51m 19s
	6	99.83%	5919	273	4460	8h 40m 11s	100.00%	697	71	404		50m 58s	100.00%	465	95	287		47m 53s
	8	99.98%	1331	212	1158	4h 39m 58s	100.00%	293	71	233		1h 10m 5s	100.00%	201	67	151		56m 12s
	10	100.00%	428	94	427	4h 45m 30s	100.00%	211	55	188		2h 4m 27s	100.00%	121	50	91		1h 16m 29s
0	4	100.00%	20631	296	16611	>13h	100.00%	1424	58	885		1h 6m 30s	100.00%	911	92	502		**37m 58s**
	6	100.00%	6093	296	4563	9h 8m 47s	100.00%	632	72	371		50m 37s	100.00%	403	85	247		38m 26s
	8	100.00%	1919	211	1567	6h 15m 29s	100.00%	378	79	287		1h 18m 16s	100.00%	174	65	128		48m 20s
	10	100.00%	402	93	401	4h 19m 3s	100.00%	180	56	154		1h 35m 56s	100.00%	82	38	63		50m 51s

(cf. columns |C|). Consequently, partitions tend to be more complex, requiring both forward and backward analyses. Since the backward analysis tends to dominate the running time, more partitions generally increase the running time (when comparing configurations with similar coverage). Based on our experience, the optimal budget largely depends on the analyzed model.

RQ6: Leveraging Multiple CPUs. To evaluate the effect of parallelizing the analysis using multiple cores, we re-ran the analyses of RQ5 on 4 CPU cores instead of 12. Table 6 shows these results. We observe the most significant increase in running time for 4 cores for the BOXES domain. On average, the running time increases by a factor of 2.6. On the other hand, for the SYMBOLIC and DEEPPOLY domains, the running time with 4 cores increases less drastically, on average by a factor of 1.6 and 2, respectively. This is again explained by the increased precision of the forward analysis; fewer partitions require a backward pass, where parallelization is most effective.

Note that the differences between the columns |C| and |F| of Table 5 and Table 6 are due to random choices that happen during the analysis, i.e., mainly the order in which (continuous) non-sensitive features are listed, which in turn dictates the order in which the partitioning happens

Table 7. Comparison of Different Analysis Configurations (Japanese Credit Screening) — 24 vCPUs

L	U	BOXES					SYMBOLIC					DEEPPOLY				
		INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME	INPUT	|C|	|F|		TIME
0.5	4	15.28%	36	0	0	7s	58.33%	120	7	34	3m 32s	69.79%	75	10	27	2m 43s
	6	17.01%	39	6	7	49s	69.10%	80	21	40	4m 19s	80.56%	138	26	65	12m 27s
	8	51.39%	92	30	86	12m 27s	82.64%	96	32	76	14m 13s	91.32%	89	36	61	13m 33s
	10	79.86%	89	34	89	29m 41s	93.06%	91	37	83	47m 1s	96.88%	73	33	52	30m
0.25	4	59.09%	1320	21	433	57m 33s	95.94%	656	42	340	32m 38s	98.26%	488	65	272	14m 11s
	6	83.77%	1600	80	1070	1h 6m 58s	98.68%	516	61	287	18m 6s	99.70%	286	77	182	13m 14s
	8	96.07%	1148	141	969	2h 41m 1s	99.72%	260	58	207	28m 57s	99.98%	241	70	175	29m 27s
	10	99.54%	409	93	403	1h 38m 38s	99.98%	213	50	189	1h 16m 11s	100.00%	88	42	68	20m 25s
0.125	4	97.13%	12449	203	9519	3h 59m 27s	99.99%	1101	59	685	1h 2m 58s	99.99%	892	86	493	18m 4s
	6	99.83%	4198	266	3234	2h 31m 54s	100.00%	759	73	461	51m 28s	100.00%	563	108	344	40m 35s
	8	99.98%	1741	217	1488	2h 16m 27s	100.00%	308	67	242	33m 14s	100.00%	230	67	167	22m 36s
	10	100.00%	582	97	564	2h 16m 13s	100.00%	180	56	154	1h 5m 59s	100.00%	80	39	62	30m 18s
0	4	100.00%	16018	288	12964	5h 3m 18s	100.00%	1883	63	1196	1h 52m 25s	100.00%	804	90	442	**19m 47s**
	6	100.00%	4675	279	3503	3h 2m 30s	100.00%	632	71	371	38m 3s	100.00%	302	75	189	19m 51s
	8	100.00%	1609	217	1382	2h 7m 9s	100.00%	326	67	252	1h 12s	100.00%	194	68	148	26m 9s
	10	100.00%	463	99	460	2h 12m 12s	100.00%	217	55	192	1h 13m 55s	100.00%	130	48	98	50m 10s

during the analysis. As a consequence, in some cases the analysis finds fewer (resp. more) activation patterns to analyze and thus might complete a little faster (resp. slower). Finding a good criterion for identifying an optimal partitioning strategy for the analysis is left for future work.

Table 7 shows the results of the same experiment on 24 vCPUs.

12 RELATED WORK

Significant progress has been made on testing and verifying machine-learning models. We focus on fairness, safety, and robustness properties in the following, especially of deep neural networks.

Fairness Criteria. There are countless fairness definitions in the literature. In this paper, we focus on dependency fairness (originally called causal fairness [Galhotra et al. 2017]) and compare here with the most popular and related fairness notions.

Demographic parity [Feldman et al. 2015] (or group fairness) is the most common non-causal notion of fairness. It states that individuals with different values of sensitive features, hence belonging to different groups, should have the same probability of being predicted to the positive class. For example, a loan system satisfies group fairness with respect to gender if male and female applicants have equal probability of getting loans. If unsatisfied, this notion is also referred to as *disparate impact*. Our notion of fairness is stronger, as it imposes fairness on every pair of individuals that differ only in sensitive features. A classifier that satisfies group fairness does not necessarily satisfy dependency fairness, because there may still exist pairs of individuals on which the classifier exhibits bias.

Another group-based notion of fairness is *equality of opportunity* [Hardt et al. 2016]. It states that *qualified* individuals with different values of sensitive features should have equal probability of being predicted to the positive class. For a loan system, this means that male and female applicants who are qualified to receive loans should have an equal chance of being approved. By imposing fairness on every qualified pair of individuals that differ only in sensitive features, we can generalize dependency fairness to also concern both prediction and actual results. We can then adapt our technique to consider only the part of the input space that includes qualified individuals.

Other causal notions of fairness [Chiappa 2019; Kilbertus et al. 2017; Kusner et al. 2017; Nabi and Shpitser 2018, etc.] require additional knowledge in the form of a *causal model* [Pearl 2009]. A causal model can drive the choice of the sensitive input(s) for our analysis.

Testing and Verifying Fairness. Galhotra et al. [Galhotra et al. 2017] proposed an approach, Themis, that allows efficient fairness testing of software. Udeshi et al. [Udeshi et al. 2018] designed an automated and directed testing technique to generate discriminatory inputs for machine-learning models. Tramer et al. [Tramèr et al. 2017] introduced the unwarranted-associations framework and instantiated it in FairTest. In contrast, our technique provides formal fairness guarantees.

Bastani et al. [Bastani et al. 2019] used adaptive concentration inequalities to design a scalable sampling technique for providing probabilistic fairness guarantees for machine-learning models. As mentioned in the Introduction, our approach differs in that it gives definite (instead of probabilistic) guarantees. However, it might exclude partitions for which the analysis is not exact.

Albarghouthi et al. [Albarghouthi et al. 2017b] encoded fairness problems as probabilistic program properties and developed an SMT-based technique for verifying fairness of decision-making programs. As discussed in the Introduction, this technique has been shown to scale only up to neural networks with at most 3 inputs and a single hidden layer with at most 2 nodes. In contrast, our approach is designed to be perfectly parallel, and thus, is significantly more scalable and can analyze neural networks with hundreds (or, in some cases, even thousands) of hidden nodes.

A recent technique [Ruoss et al. 2020] certifies individual fairness of neural networks, which is a local property that coincides with robustness within a particular distance metric. In particular, individual fairness dictates that similar individuals should be treated similarly. Our approach, however, targets certification of neural networks for the global property of dependency fairness.

For certain biased decision-making programs, the program repair technique proposed by Albarghouthi et al. [Albarghouthi et al. 2017a] can be used to repair their bias. Albarghouthi and Vinitsky [Albarghouthi and Vinitsky 2019] further introduced fairness-aware programming, where programmers can specify fairness properties in their code for runtime checking.

Robustness of Deep Neural Networks. Robustness is a desirable property for traditional software [Chaudhuri et al. 2012; Goubault and Putot 2013; Majumdar and Saha 2009], especially control systems. Deep neural networks are also expected to be robust. However, research has shown that deep neural networks are not robust to small perturbations of their inputs [Szegedy et al. 2014] and can even be easily fooled [Nguyen et al. 2015]. Subtle imperceptible perturbations of inputs, known as adversarial examples, can change their prediction results. Various algorithms [Carlini and Wagner 2017b; Goodfellow et al. 2015; Madry et al. 2018; Tabacof and Valle 2016; Zhang et al. 2020] have been proposed that can effectively find adversarial examples. Research on developing defense mechanisms against adversarial examples [Athalye et al. 2018; Carlini and Wagner 2016, 2017a,b; Cornelius 2019; Engstrom et al. 2018; Goodfellow et al. 2015; Huang et al. 2015; Mirman et al. 2018, 2019] is also active. Dependency fairness is a special form of robustness in the sense that neural networks are expected to be *globally* robust with respect to their sensitive features.

Testing Deep Learning Systems. Multiple frameworks have been proposed to test the robustness of deep learning systems. Pei et al. [Pei et al. 2017] proposed the first whitebox framework for testing such systems. They used neuron coverage to measure the adequacy of test inputs. Sun et al. [Sun et al. 2018] presented the first concolic-testing [Godefroid et al. 2005; Sen et al. 2005] approach for neural networks. Tian et al. [Tian et al. 2018] and Zhang et al. [Zhang et al. 2018] proposed frameworks for testing autonomous driving systems. Gopinath et al. [Gopinath et al. 2018] used symbolic execution [Clarke 1976; King 1976]. Odena et al. [Odena et al. 2019] were the first to develop coverage-guided fuzzing for neural networks. Zhang et al. [Zhang et al. 2020] proposed a blackbox-fuzzing technique to test their robustness.

Formal Verification of Deep Neural Networks. Formal verification of deep neural networks has mainly focused on safety properties. However, the scalability of such techniques for verifying

large real-world neural networks is limited. Early work [Pulina and Tacchella 2010] applied abstract interpretation to verify a neural network with six neurons. Recent work [Gehr et al. 2018; Huang et al. 2017; Katz et al. 2017; Singh et al. 2019; Wang et al. 2018] significantly improves scalability. Huang et al. [Huang et al. 2017] proposed a framework that can verify local robustness of neural networks based on SMT techniques [Barrett and Tinelli 2018]. Katz et al. [Katz et al. 2017] developed an efficient SMT solver for neural networks with ReLU activation functions. Gehr et al. [Gehr et al. 2018] traded precision for scalability and proposed a sound abstract interpreter that can prove local robustness of realistic deep neural networks. Singh et al. [Singh et al. 2019] proposed the DEEPPOLY domain for certifying robustness of neural networks. Wang et al. [Wang et al. 2018] are the first to use symbolic interval arithmetic to prove security properties of neural networks.

13 CONCLUSION AND FUTURE WORK

We have presented a *novel, automated, perfectly parallel static analysis* for certifying fairness of feed-forward neural networks used for classification of tabular data — a problem of ever-increasing real-world importance nowadays. Our approach provides *definite fairness guarantees* for the analyzed input space and it is the first in this area that is *configurable in terms of scalability and precision*. The analysis can thus support a wide range of use cases throughout the development lifecycle of neural networks: ranging from quick sanity checks during development to formal fairness audits before deployments. We have rigorously formalized the approach and proved its soundness, and we demonstrated its effectiveness in practice with an extensive experimental evaluation.

In future work, we plan on integrating strategies in LIBRA for making the parameter configuration of U and L automatic during the analysis. For instance, by starting the analysis with a low U (resp. large L) and gradually increasing (resp. decreasing) it only on the input space partitions on which it is necessary. We also plan on supporting other encodings for categorical features than one-hot encoding, such as entity embeddings [Guo and Berkhahn 2016], which have become more and more popular in recent years. Our approach will also automatically benefit from future advances in the design of abstract domains for analyzing neural networks. Vice versa, we believe and hope that future work can also build on our approach. For instance, other tools could feed on the results of our analysis and, say, provide weaker fairness guarantees (e.g., probabilistic) for the input partitions that could not be certified, or repair the analyzed neural network models by eliminating bias. More generally, we believe that the design of the approach and its underlying ideas, such as that of leveraging abstract activation patterns, are potentially useful in other verification settings, e.g., proving functional properties of neural networks [Katz et al. 2017; Wang et al. 2018, etc.].

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive feedback. This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

REFERENCES

Aws Albarghouthi, Loris D'Antoni, and Samuel Drews. 2017a. Repairing Decision-Making Programs Under Uncertainty. In *CAV*. 181–200. https://doi.org/10.1007/978-3-319-63387-9_9

Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017b. FairSquare: Probabilistic Verification of Program Fairness. *PACMPL* 1, OOPSLA (2017), 80:1–80:30. https://doi.org/10.1145/3133904

Aws Albarghouthi and Samuel Vinitsky. 2019. Fairness-Aware Programming. In *FAT**. 211–219. https://doi.org/10.1145/3287560.3287588

Anish Athalye, Nicholas Carlini, and David A. Wagner. 2018. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. In *ICML (PMLR)*, Vol. 80. PMLR, 274–283.

Solon Barocas and Andrew D. Selbst. 2016. Big Data's Disparate Impact. *California Law Review* 104, 3 (2016), 671–732.

Clark W. Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*. Springer, 305–343.

Alexander I. Barvinok. 1994. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra When the Dimension is Fixed. *Mathematics of Operations Research* 19, 4 (1994), 769–779. https://doi.org/10.1287/moor.19.4.769

Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2019. Probabilistic Verification of Fairness Properties via Concentration. *PACMPL* 3, OOPSLA (2019), 118:1–118:27.

Joy Buolamwini and Timnit Gebru. 2018. Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification. In *FAT (PMLR)*, Vol. 81. PMLR, 77–91.

Nicholas Carlini and David A. Wagner. 2016. Defensive Distillation is Not Robust to Adversarial Examples. *CoRR* abs/1607.04311 (2016).

Nicholas Carlini and David A. Wagner. 2017a. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. In *AISec@CCS*. ACM, 3–14.

Nicholas Carlini and David A. Wagner. 2017b. Towards Evaluating the Robustness of Neural Networks. In *S&P*. IEEE Computer Society, 39–57.

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2012. Continuity and Robustness of Programs. *Commun. ACM* 55, 8 (2012), 107–115. https://doi.org/10.1145/2240236.2240262

Liqian Chen, Antoine Miné, and Patrick Cousot. 2008. A Sound Floating-Point Polyhedra Abstract Domain. In *APLAS*. 3–18. https://doi.org/10.1007/978-3-540-89330-1_2

Silvia Chiappa. 2019. Path-Specific Counterfactual Fairness. In *AAAI*. 7801–7808. https://doi.org/10.1609/aaai.v33i01.33017801

Lori A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *TSE* 2 (1976), 215–222. Issue 3.

Cory Cornelius. 2019. The Efficacy of SHIELD Under Different Threat Models. *CoRR* abs/1902.00541 (2019).

Patrick Cousot. 2002. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science* 277, 1-2 (2002), 47–103. https://doi.org/10.1016/S0304-3975(00)00313-3

Patrick Cousot and Radhia Cousot. 1976. Static Determination of Dynamic Properties of Programs. In *Second International Symposium on Programming*. 106–130.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. 269–282. https://doi.org/10.1145/567752.567778

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*. 84–96. https://doi.org/10.1145/512760.512770

Anupam Datta, Matthew Fredrikson, Gihyuk Ko, Piotr Mardziel, and Shayak Sen. 2017. Use Privacy in Data-Driven Systems: Theory and Experiments with Machine Learnt Programs. In *CCS*. 1193–1210. https://doi.org/10.1145/3133956.3134097

Logan Engstrom, Andrew Ilyas, and Anish Athalye. 2018. Evaluating and Understanding the Robustness of Adversarial Logit Pairing. *CoRR* abs/1807.10272 (2018).

Michael Feldman, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and Removing Disparate Impact. In *KDD*. ACM, 259–268.

Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness Testing: Testing Software for Discrimination. In *FSE*. 498–510. https://doi.org/10.1145/3106237.3106277

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *S & P*. 3–18. https://doi.org/10.1109/SP.2018.00058

Khalil Ghorbal, Eric Goubault, and Sylvie Putot. 2009. The Zonotope Abstract Domain Taylor1+. In *CAV*. 627–633. https://doi.org/10.1007/978-3-642-02658-4_47

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI*. ACM, 213–223.

Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *ICLR*. http://arxiv.org/abs/1412.6572

Divya Gopinath, Kaiyuan Wang, Mengshi Zhang, Corina S. Pasareanu, and Sarfraz Khurshid. 2018. Symbolic Execution for Deep Neural Networks. *CoRR* abs/1807.10439 (2018).

Eric Goubault and Sylvie Putot. 2013. Robustness Analysis of Finite Precision Implementations. In *APLAS*. 50–57. https://doi.org/10.1007/978-3-319-03542-0_4

Nina Grgić-Hlača, Muhammad Bilal Zafar, Krishna P. Gummadi, and Adrian Weller. 2016. The Case for Process Fairness in Learning: Feature Selection for Fair Decision Making. In *NIPS 2016 ML and the Law*.

Cheng Guo and Felix Berkhahn. 2016. Entity Embeddings of Categorical Variables. *CoRR* abs/1604.06737 (2016).

Boris Hanin and David Rolnick. 2019. Deep ReLU Networks Have Surprisingly Few Activation Patterns. In *NIPS*. Curran Associates, Inc., 359–368. http://papers.nips.cc/paper/8328-deep-relu-networks-have-surprisingly-few-activation-

patterns.pdf
Moritz Hardt, Eric Price, and Nati Srebro. 2016. Equality of Opportunity in Supervised Learning. In *NIPS*. 3315–3323.
Ruitong Huang, Bing Xu, Dale Schuurmans, and Csaba Szepesvári. 2015. Learning with a Strong Adversary. *CoRR* abs/1511.03034 (2015). http://arxiv.org/abs/1511.03034
Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *CAV*. 3–29. https://doi.org/10.1007/978-3-319-63387-9_1
Bertrand Jeannet and Antoine Miné. 2009. APRON: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*. 661–667. https://doi.org/10.1007/978-3-642-02658-4_52
Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *CAV*. 97–117. https://doi.org/10.1007/978-3-319-63387-9_5
Matthew Kay, Cynthia Matuszek, and Sean A. Munson. 2015. Unequal Representation and Gender Stereotypes in Image Search Results for Occupations. In *CHI*. ACM, 3819–3828.
Niki Kilbertus, Mateo Rojas-Carulla, Giambattista Parascandolo, Moritz Hardt, Dominik Janzing, and Bernhard Schölkopf. 2017. Avoiding Discrimination Through Causal Reasoning. In *NIPS*. 656–666.
James C. King. 1976. Symbolic Execution and Program Testing. *CACM* 19 (1976), 385–394. Issue 7.
Matt Kusner, Joshua Loftus, Chris Russell, and Ricardo Silva. 2017. Counterfactual Fairness. In *NIPS*. 4069–4079.
Jeff Larson, Surya Mattu, Lauren Kirchner, and Julia Angwin. 2016. How We Analyzed the COMPAS Recidivism Algorithm. https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm.
Jianlin Li, Jiangchao Liu, Pengfei Yang, Liqian Chen, Xiaowei Huang, and Lijun Zhang. 2019. Analyzing Deep Neural Networks with Symbolic Propagation: Towards Higher Precision and Faster Verification. In *SAS*. 296–319. https://doi.org/10.1007/978-3-030-32304-2_15
Kristian Lum and William Isaac. 2016. To Predict and Serve? *Significance* 13 (2016), 14–19. Issue 5.
Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *ICLR*. OpenReview.net.
Rupak Majumdar and Indranil Saha. 2009. Symbolic Robustness Analysis. In *RTSS*. 355–363. https://doi.org/10.1109/RTSS.2009.17
Padala Manisha and Sujit Gujar. 2020. FNNC: Achieving Fairness Through Neural Networks. In *IJCAI*. 2277–2283.
Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. 2019. A Survey on Bias and Fairness in Machine Learning. *CoRR* abs/1908.09635 (2019).
Antoine Miné. 2004. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *ESOP*. 3–17. https://doi.org/10.1007/978-3-540-24725-8_2
Antoine Miné. 2006a. Symbolic Methods to Enhance the Precision of Numerical Abstract Domains. In *VMCAI*. 348–363. https://doi.org/10.1007/11609773_23
Antoine Miné. 2006b. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation* 19, 1 (2006), 31–100. https://doi.org/10.1007/s10990-006-8609-1
Matthew Mirman, Timon Gehr, and Martin T. Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *ICML*. 3575–3583.
Matthew Mirman, Gagandeep Singh, and Martin T. Vechev. 2019. A Provable Defense for Deep Residual Networks. *CoRR* abs/1903.12519 (2019).
Razieh Nabi and Ilya Shpitser. 2018. Fair Inference on Outcomes. In *AAAI*. AAAI Press.
Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *ICML*. 807–814.
Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. 2015. Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In *CVPR*. 427–436. https://doi.org/10.1109/CVPR.2015.7298640
Ziad Obermeyer, Brian Powers, Christine Vogeli, and Sendhil Mullainathan. 2019. Dissecting Racial Bias in an Algorithm Used to Manage the Health of Populations. *Science* 366 (2019), 447–453. Issue 6464.
Augustus Odena, Catherine Olsson, David Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *ICML (PMLR)*, Vol. 97. PMLR, 4901–4911.
Judea Pearl. 2009. *Causality: Models, Reasoning and Inference*. Cambridge University Press.
Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *SOSP*. 1–18. https://doi.org/10.1145/3132747.3132785
Luca Pulina and Armando Tacchella. 2010. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *CAV*. 243–257. https://doi.org/10.1007/978-3-642-14295-6_24
Anian Ruoss, Mislav Balunovic, Marc Fischer, and Martin T. Vechev. 2020. Learning Certified Individually Fair Representations. *CoRR* abs/2002.10312 (2020).
Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE*. ACM, 263–272.
Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An Abstract Domain for Certifying Neural Networks. *PACMPL* 3, POPL (2019), 41:1–41:30. https://doi.org/10.1145/3290354

Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *ASE*. ACM, 109–119.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing Properties of Neural Networks. In *ICLR*. http://arxiv.org/abs/1312.6199

Pedro Tabacof and Eduardo Valle. 2016. Exploring the Space of Adversarial Images. In *IJCNN*. 426–433. https://doi.org/10.1109/IJCNN.2016.7727230

Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *ICSE*. ACM, 303–314.

Florian Tramèr, Vaggelis Atlidakis, Roxana Geambasu, Daniel J. Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2017. FairTest: Discovering Unwarranted Associations in Data-Driven Applications. In *EuroS&P*. IEEE, 401–416.

Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. 2020. Verification of Deep Convolutional Neural Networks Using ImageStars. In *CAV*. 18–42.

Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated Directed Fairness Testing. In *ASE*. ACM, 98–108.

Caterina Urban and Peter Müller. 2018. An Abstract Interpretation Framework for Input Data Usage. In *ESOP*. 683–710. https://doi.org/10.1007/978-3-319-89884-1_24

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks Using Symbolic Intervals. In *Security*. USENIX, 1599–1614.

Mikhail Yurochkin, Amanda Bower, and Yuekai Sun. 2020. Training Individually Fair ML Models with Sensitive Subspace Robustness. In *ICLR*.

Fuyuan Zhang, Sankalan Pal Chowdhury, and Maria Christakis. 2020. DeepSearch: A Simple and Effective Blackbox Attack for Deep Neural Networks. In *ESEC/FSE*. ACM. To appear.

Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *ASE*. ACM, 132–142.

[c22] Marco Campion, **CU**, Mila Dalla Preda, Roberto Giacobazzi

A Formal Framework to Measure the Incompleteness of Abstract Interpretations

In 30th Static Analysis Symposium (SAS 2023)

https://inria.hal.science/hal-04249990

A Formal Framework to Measure the Incompleteness of Abstract Interpretations

Marco Campion[1], Caterina Urban[1], Mila Dalla Preda[2], and Roberto Giacobazzi[3]

[1] INRIA & École Normale Supérieure | Université PSL, Paris, France
{marco.campion,caterina.urban}@inria.fr
[2] University of Verona, Department of Computer Science, Verona, Italy
mila.dallapreda@univr.it
[3] University of Arizona, Department of Computer Science, Tucson, Arizona, USA
giacobazzi@arizona.edu

Abstract. In program analysis by abstract interpretation, backward-completeness represents no loss of precision between the result of the analysis and the abstraction of the concrete execution, while forward-completeness stands for no imprecision between the concretization of the analysis result and the concrete execution. Program analyzers satisfying one of the two properties (or both) are considered precise. Regrettably, as for all approximation methods, the presence of false-alarms is most of the time unavoidable and therefore we need to deal somehow with incompleteness of both. To this end, a new property called partial completeness has recently been formalized as a relaxation of backward-completeness allowing a limited amount of imprecision measured by quasi-metrics. However, the use of quasi-metrics enforces distance functions to adhere precisely the abstract domain ordering, thus not suitable to be used to weaken the forward-completeness property which considers also abstract domains that are not necessarily based on Galois Connections. In this paper, we formalize a weaker form of quasi-metric, called pre-metric, which can be defined on all domains equipped with a pre-order relation. We show how this newly defined notion of pre-metric allows us to derive other pre-metrics on other domains by exploiting the concretization and, when available, the abstraction maps, according to the information and the corresponding level of approximation that we want to measure. Finally, by exploiting pre-metrics as our imprecision meter, we introduce the partial forward/backward-completeness properties.

Keywords: Abstract Interpretation · Partial Completeness · Completeness · Program Analysis · Distances

1 Introduction

The theory of Abstract Interpretation introduced by Cousot and Cousot [20,21,22], is a general theory for the approximation of formal program semantics based on a simple but striking idea that extracting properties of programs' execution

means over-approximating their semantics. It is an invaluable framework that helps programmers design sound-by-construction program analysis tools as it makes possible to express mathematically the link between the output of a practical, approximated analysis, also called abstract semantics, and the original, uncomputable program semantics, also called concrete semantics.

The abstract interpretation of a program P consists of an abstract domain of properties of interest $\mathcal{A}$ ordered by a partial-order $\leq_{\mathcal{A}}$, a concretization map γ and an abstract interpreter $[\![\cdot]\!]_{\mathcal{A}}$, designed for the language used to specify P and on the abstract domain $\mathcal{A}$. Let $[\![P]\!]S$ be the result of the concrete (collecting) program semantics on a set of concrete inputs S. *Soundness* means that for all possible abstract inputs $S^\sharp \in \mathcal{A}$ it holds $[\![P]\!]\gamma(S^\sharp) \subseteq \gamma([\![P]\!]_{\mathcal{A}} S^\sharp)$. Furthermore, when $[\![P]\!]_{\mathcal{A}}$ also satisfies $[\![P]\!]\gamma(S^\sharp) = \gamma([\![P]\!]_{\mathcal{A}} S^\sharp)$ then $[\![P]\!]_{\mathcal{A}}$ is said to be *forward-complete* [32], while if the equation holds for a given input $S^\sharp \in \mathcal{A}$, then it is *locally* forward-complete at $S^\sharp$. In abstract interpretation forward-completeness intuitively encodes the greatest achievable precision for an abstract interpreter $[\![\cdot]\!]_{\mathcal{A}}$ applied on a program P, meaning that $[\![P]\!]_{\mathcal{A}} S^\sharp$ exactly matches the concrete result of the concrete counterpart $[\![P]\!]\gamma(S^\sharp)$. When the abstraction $\mathcal{A}$ also admits a Galois Connection (GC) with the concrete domain through an abstraction map α, then $[\![P]\!]_{\mathcal{A}}$ is said to be *backward-complete* when $\alpha([\![P]\!]S) = [\![P]\!]_{\mathcal{A}}\alpha(S)$ holds for all possible concrete set of inputs S, while *locally* backward-complete [5,7] at S if it holds for the input S. The backward-completeness property encodes an optimal behavior of the abstract interpreter $[\![\cdot]\!]_{\mathcal{A}}$ with respect to the abstraction in $\mathcal{A}$ of the concrete behavior $[\![P]\!]S$. Forward- and backward-completeness and their local versions are both highly desirable properties in program analysis for verifying safety properties of programs (also called correctness properties) [21,33,29,39]. Unfortunately, it is well known that whenever a non-trivial abstract domain is used, the analysis will be necessarily (locally) forward/backward-incomplete, meaning that false alarms or spurious counterexamples will arise also for correct programs [29,11]. In fact, forward/backward-completeness and their local definitions in program analysis are extremely hard, if not even impossible, to achieve [33,11]. For this reason, instead of trying to reach forward/backward-completeness, we need to deal with incompleteness of both and therefore with imprecision [27].

In this direction, the notion of *partial completeness* has been introduced in [11] in order to weaken the equality requirement of the local backward-completeness property. Partial completeness allows a limited amount of incompleteness and this amount is measured by quasi-metrics $\delta_{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \to \mathbb{R}^{\infty}_{\geq 0} \cup \{\bot\}$ (where the symbol $\bot$ means undefined) compatible with the underlying abstract domain partial-ordering. More specifically, for a distance function $\delta_{\mathcal{A}}$ being a quasi-metric $\mathcal{A}$-compatible means satisfying for all $a_1, a_2, a_3 \in \mathcal{A}$: (i) the partial-ordering $\leq_{\mathcal{A}}$: $a_1 \leq_{\mathcal{A}} a_2 \;\Leftrightarrow\; \delta_{\mathcal{A}}(a_1, a_2) \neq \bot$; (ii) identity of indiscernibles: $a_1 = a_2 \;\Leftrightarrow\; \delta(a_1, a_2) = 0$; and (iii) the weak triangle inequality, namely, the triangle inequality only along chains $a_1 \leq_{\mathcal{A}} a_2 \leq_{\mathcal{A}} a_3$. So for instance, consider the intervals abstract domain [19] $\mathsf{Int} \stackrel{def}{=} \{[a, b] \mid a, b \in \mathbb{Z}^*,\ a \leq b\} \cup \{\bot_{\mathsf{Int}}\}$, where $\mathbb{Z}^* \stackrel{def}{=} \mathbb{Z} \cup \{-\infty, +\infty\}$, endowed with the standard ordering $\leq_{\mathsf{Int}}$ induced

by the interval containment. We can consider as quasi-metric Int-compatible the distance $\delta^{\mathfrak{w}}_{\mathsf{Int}}$ that counts how many more integer values has one interval with respect to another comparable interval. For instance, $\delta^{\mathfrak{w}}_{\mathsf{Int}}([0,0],[0,5]) = 5$, $\delta^{\mathfrak{w}}_{\mathsf{Int}}([0,+\infty],[-2,+\infty]) = 2$, while $\delta^{\mathfrak{w}}_{\mathsf{Int}}([0,5],[0,0]) = \bot$ as $[0,5] \not\leq_{\mathsf{Int}} [0,0]$. The analysis $[\![P]\!]_{\mathcal{A}}\alpha(S)$ on a program P with input S is said to be ε-partial complete at input S for an amount $\varepsilon \in \mathbb{R}^{\infty}_{\geq 0}$ whenever $\delta_{\mathcal{A}}(\alpha([\![P]\!]S), [\![P]\!]_{\mathcal{A}}\alpha(S)) \leq \varepsilon$ holds, namely, the distance between the abstraction of the concrete execution and the result of the abstract interpreter is at maximum ε. In this setting, requiring 0-partial completeness at S corresponds to require local backward-completeness at S.

Main Contribution. In this paper we generalize the partial completeness property in order to be able to weaken both the local backward-completeness property in presence of a GC, *and* the local forward-completeness property in case only the concretization function γ is available. In this last scenario, as we may need to define distances on concrete domains, a weakening of the definition of quasi-metrics $\mathcal{A}$-compatible is necessary. This is because, in the original formalization [11], the definition of quasi-metric $\mathcal{A}$-compatible is specifically tailored for the structure of abstract domains and their relative partial-ordering: the axiom (i) forces the quasi-metric to return a value different from $\bot$ only if the two elements are comparable according to $\leq_{\mathcal{A}}$, namely $\delta_{\mathcal{A}}$ induces the partial-order $\leq_{\mathcal{A}}$. This is in fact not necessary: as our aim is to measure the incompleteness of an abstract interpreter with respect to the concrete execution, these two results are guaranteed to be comparable by soundness, therefore the distance may even return values on non-comparable elements as long as it is defined on all comparable ones. Moreover, the identity of indiscernibles axiom (ii) requires the quasi-metric to be precise enough to recognize equal elements since it constraints the distance to return zero whenever the two elements are equal. This is a too strong requirement especially in the scenario where only γ is available and we have to define a distance on the concrete domain where elements contain more information than what we are interested in for measuring the incompleteness. For instance, by considering the concrete domain $\wp(\mathbb{Z}^n)$ where n is the number of variables used in a program and elements in $S \in \wp(\mathbb{Z}^n)$ are program states, we might need a distance function $\delta_{\wp(\mathbb{Z}^n)}$ that measures the imprecision of certain variables only, say x and y. A possible estimate of this imprecision could be done by calculating the volume of their abstraction into the intervals abstract domain, namely, the area of the rectangle abstracting the values of x and y.

To this end, in Section 3 we reason on the *weakest* axioms that a distance function should meet so that it can be used to measure the local forward/backward-incompleteness. We just require a relaxed version of the identity of indiscernibles axiom (only the $\Rightarrow$ direction) and a condition on chains. As domains may not be complete lattices or partial-orders, such as the convex polyhedra abstract domain [24], we only require one of the weakest form of ordering relation known as a pre-order. The resulting distance function will be called *pre-metric* $\preceq_D$*-compatible* where D is a pre-ordered set according to the pre-order $\preceq_D$. We will provide several useful examples of pre-metrics compatible to generic pre-ordered

sets, as well as well-known numerical abstract domains, that can be used in practice. In Section 4 we show how this newly defined notion of pre-metric $\preceq_D$-compatible allows us to derive other pre-metrics from one domain to another by exploiting the concretization γ or, when available, the abstraction map α. Finally, in Section 5 we define the new properties *partial backward-completeness* and *partial forward-completeness* using pre-metrics compatible with the underlying domain ordering. We show that, when a certain condition on the precision of the pre-metric is met, then we can characterize the local forward/backward-completeness as the 0-partial forward/backward-completeness. The proposed framework is general enough to be instantiated by most known metrics for abstract interpretation [25,36,42,13,11]. Since imprecision, i.e., incompleteness, is unavoidable in program analysis, our ambition is to help abstract interpretation designers in defining distances able to measure the imprecision they *want to track* regardless of the domain on which they want to define the distance, hence providing the appropriate tools to fully control the imprecision propagation.

2 Background

Orderings. Given two sets S and T, $\wp(S)$ denotes the powerset of S, $\varnothing$ is the empty set, $S \subseteq T$ denotes sets inclusion, $|S|$ denotes the cardinality where S is finite if $|S| < \omega$, countably infinite if $|S| = \omega$, countable if $|S| \leq \omega$. A binary relation $\sim$ over a set S is a subset of the Cartesian product $\sim \subseteq S \times S$. We will emphasize the set S on which a binary relation $\sim$ is defined by the subscript $\sim_S$ except for the straightforward equivalence relation $=$ unless it has a different definition. We denote with $\mathbb{Z}$ and $\mathbb{R}$ the sets of all, respectively, integer and real numbers. We will use subscripts in order to limit their range, while the superscript symbol ∞ denotes the inclusion of the infinite symbol. For example, $\mathbb{R}^{\infty}_{\geq 0}$ denotes the set of all non-negative real numbers together with the symbol ∞ such that, for all $\varepsilon \in \mathbb{R}_{\geq 0}$, $\varepsilon < \infty$.

A binary relation $\preceq_L \in \wp(L \times L)$ is a *pre-order* iff it is reflexive ($\forall l \in L.\ l \preceq_L l$) and transitive ($\forall l_1, l_2, l_3 \in L.\ l_1 \preceq_L l_2 \wedge l_2 \preceq_L l_3 \Rightarrow l_1 \preceq_L l_3$). A set L endowed with a pre-order relation $\preceq_L$ is called a *pre-ordered set*, and it is denoted by $(L, \preceq_L)$. Furthermore, if $\preceq_L$ is anti-symmetric ($\forall l_1, l_2 \in L.\ l_1 \preceq_L l_2 \wedge l_2 \preceq_L l_1 \Rightarrow l_1 = l_2$) then it is a *partial-order* and the pair $(L, \preceq_L)$ is called a *partially-ordered set*. Clearly, every partially-ordered set is also a pre-ordered set. A subset $Y \subseteq L$ of a pre-ordered set $(L, \preceq_L)$ is a *chain* iff for all $y_1, y_2 \in Y$, $y_1 \preceq_L y_2$ or $y_2 \preceq_L y_1$.

Measures and Distances. A σ-algebra on a set X is a collection of subsets of X that includes X itself, it is closed under complement and it is closed under countable unions. The definition implies that it also includes the empty set $\varnothing$ and that it is closed under countable intersections. Consider a σ-algebra A over X. The tuple (X, A) is called a *measurable space*.

Definition 1 (Measure). *A function $\mu : A \to \mathbb{R}^{\infty}_{\geq 0}$ is called a measure iff it satisfies the following properties:*

(1) *non-negativity:* $\forall S \in A.\ \mu(S) \geq 0$*;*
(2) *null empty set:* $\mu(\varnothing) = 0$*;*
(3) *countable additivity: if* $S_i \in A$ *is a countable sequence of disjoint sets, then* $\mu(\bigcup_i S_i) = \sum_i \mu(S_i)$.

The triple (X, A, μ) *is called a measure space.* ■

A *metric* is a function that defines a distance between pairs of elements of a set S. Formally:

Definition 2 (Metric). *A metric on a non-empty set* S *is a map* $\delta_S : S \times S \to \mathbb{R}_{\geq 0}$ *that* $\forall x, y, z \in S$ *satisfies:*

(1) *identity of indiscernibles:* $x = y \Leftrightarrow \delta_S(x, y) = 0$*;*
(2) *symmetry:* $\delta_S(x, y) = \delta_S(y, x)$*;*
(3) *triangle inequality:* $\delta_S(x, z) \leq \delta_S(x, y) + \delta_S(y, z)$.

A set provided with a metric is called a metric space. ■

A function $\delta_S : S \times S \to \mathbb{R}_{\geq 0}$ satisfying all axioms of Definition 2 except for symmetry, is called a *quasi-metric*, while if δ_S does not satisfy the $\Leftarrow$ direction of the identity of indiscernibles axiom then it is called a *pseudo-metric*. A pseudoquasi-metric relaxes both the indiscernibility axiom and the symmetry axiom of a metric. δ_S is said to be a *pre-metric* if it satisfies only the $\Rightarrow$ implication of the identity of indiscernibility axiom (symmetry and triangle inequality may not hold).

Abstract Interpretation. We consider here the abstract interpretation framework as defined in [22] and based on the correspondence between a domain of concrete or exact properties $\mathcal{C}$ and a domain of abstract or approximate properties $\mathcal{A}$. Concrete and abstract domains are assumed to be at least pre-ordered sets, respectively $(\mathcal{C}, \preceq_{\mathcal{C}})$ and $(\mathcal{A}, \preceq_{\mathcal{A}})$, and be related by a monotone *concretization* function $\gamma : \mathcal{A} \to \mathcal{C}$. Furthermore, when they enjoy a *Galois Connection* (GC) through a monotone *abstraction* function $\alpha : \mathcal{C} \to \mathcal{A}$, denoted by the symbols $(\mathcal{C}, \preceq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_{\mathcal{A}})$, then for all $a \in \mathcal{A}$ and $c \in \mathcal{C}$: $\alpha(c) \preceq_{\mathcal{A}} a \ \Leftrightarrow\ c \preceq_{\mathcal{C}} \gamma(a)$. A GC is a *Galois Insertion* (GI), denoted by $(\mathcal{C}, \preceq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\!\!\rightarrow}} (\mathcal{A}, \preceq_{\mathcal{A}})$, when it holds $\alpha \circ \gamma = id$, where $\circ$ denotes functions composition and id is the identity function. A concrete element $c \in \mathcal{C}$ is said to be *exactly representable* in the abstract domain $\mathcal{A}$ when $\gamma(\alpha(c)) = c$.

Soundness and Completeness. Let $f_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ be a concrete monotone operator (to keep notation simple we consider unary functions) and let $f_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ be a corresponding monotone abstract operator defined on some abstraction $\mathcal{A}$. Then, $f_{\mathcal{A}}$ is a *sound* (or *correct*) approximation of $f_{\mathcal{C}}$ on $\mathcal{A}$ when for all $a \in \mathcal{A}$, $f_{\mathcal{C}}(\gamma(a)) \preceq_{\mathcal{C}} \gamma(f_{\mathcal{A}}(a))$ holds. When dealing with GCs, between all abstract functions that approximate a concrete one we can define the most precise one called *best correct approximation* (*bca* for short): $f^{\alpha}_{\mathcal{A}} \overset{def}{=} \alpha \circ f_{\mathcal{C}} \circ \gamma$. It turns out that any abstract function $f_{\mathcal{A}}$ is a correct approximation of $f_{\mathcal{C}}$ if and only if it holds $f^{\alpha}_{\mathcal{A}} \preceq_{\mathcal{A}} f_{\mathcal{A}}$ [20].

Given an abstract input $a \in \mathcal{A}$, when the concretization of $f_\mathcal{A}(a)$ matches the concrete counterpart $f_\mathcal{C}(\gamma(a))$ then $f_\mathcal{A}$ is said to be *locally forward-complete*[1] at the input $a \in \mathcal{A}$.

Definition 3 (Local forward-completeness). *Let $f_\mathcal{A} : \mathcal{A} \to \mathcal{A}$ be a sound approximation of $f_\mathcal{C} : \mathcal{C} \to \mathcal{C}$. Given an input $a \in \mathcal{A}$, $f_\mathcal{A}$ is said to be locally forward-complete at the input a, when $f_\mathcal{C}(\gamma(a)) = \gamma(f_\mathcal{A}(a))$ holds.* ■

When $\mathcal{C}$ and $\mathcal{A}$ admit a GC, then we can also define the property of *local backward-completeness* [5,7].

Definition 4 (Local backward-completeness). *Let $(\mathcal{C}, \preceq_\mathcal{C}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_\mathcal{A})$ and $f_\mathcal{A}$ be a sound approximation of $f_\mathcal{C}$. Given an input $c \in \mathcal{C}$, $f_\mathcal{A}$ is said to be locally backward-complete at the input c, when $\alpha(f_\mathcal{C}(c)) = f_\mathcal{A}(\alpha(c))$.* ■

The local forward- and backward-completeness properties are a weakening of the standard notions of forward- [32] and backward-completeness[2] [20,21,33], respectively, which require Definition 3 and 4 to hold over all possible, respectively, abstract and concrete inputs. Intuitively, when $f_\mathcal{A}$ is an abstract transfer function on $\mathcal{A}$ used in some static program analysis algorithm, local backward-completeness at input $c \in \mathcal{C}$ encodes an optimal precision for $f_\mathcal{A}$ at input $\alpha(c)$, meaning that the abstract behavior of $f_\mathcal{A}(\alpha(c))$ on $\mathcal{A}$ exactly matches the abstraction in $\mathcal{A}$ of the concrete behavior of $f_\mathcal{C}(c)$. On the other hand, if $f_\mathcal{A}$ is locally forward-complete at the abstract input $a \in \mathcal{A}$ means that $f_\mathcal{A}$ acts on the abstract input a precisely as $f_\mathcal{C}$ does on its concretization. As a remark, when $f_\mathcal{A}$ is locally forward-complete on an input $a \in \mathcal{A}$ and $(\mathcal{C}, \preceq_\mathcal{C}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\!\rightarrow}} (\mathcal{A}, \preceq_\mathcal{A})$, then $f_\mathcal{A}$ is locally backward-complete at $\gamma(a)$, namely, $f_\mathcal{A}(a)$ corresponds to the bca $f^\alpha_\mathcal{A}(a)$.

A relaxation of Definition 4 has been introduced in [11], called *partial completeness*, where quasi-metrics compatible with the underlying abstract domain are considered to measure the imprecision of $f_\mathcal{A}(\alpha(c))$ compared to $\alpha(f_\mathcal{C}(c))$.

Definition 5 (ε-Partial (backward-)completeness). *Consider the Galois Connection $(\mathcal{C}, \preceq_\mathcal{C}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_\mathcal{A})$, a sound approximation $f_\mathcal{A}$ of $f_\mathcal{C}$, a quasi-metric $\delta_\mathcal{A}$ $\mathcal{A}$-compatible, and $\varepsilon \in \mathbb{R}^\infty_{\geq 0}$. The abstract operator $f_\mathcal{A}$ is said to be an ε-partial (backward-)complete approximation of $f_\mathcal{C}$ on input $c \in \mathcal{C}$ when the following inequality holds: $\delta_\mathcal{A}(\alpha(f_\mathcal{C}(c)), f_\mathcal{A}(\alpha_\mathcal{A}(c))) \leq \varepsilon$.* ■

Establishing ε-partial (backward-)completeness at input $c \in \mathcal{C}$ of an abstract operator $f_\mathcal{A}$, means that when computing $f_\mathcal{A}(\alpha(c))$, the output result is allowed to have an imprecision limited to ε compared to the abstraction of the concrete execution at c, namely, $\alpha(f_\mathcal{C}(c))$. The meaning of the value ε depends on the quasi-metric $\mathcal{A}$-compatible chosen. Note that the ε-partial (backward-)completeness property is always considered with respect to a specified input.

[1] The term "forward-completeness" was introduced in [32] in order to distinguish it from the well known backward-completeness property requiring an abstraction function.

[2] In the standard abstract interpretation framework [20,21] dealing with GCs, the backward-completeness property is simply called completeness or exactness.

3 Distances on Orderings

The goal of this section is to set the minimum requirements that a distance function must meet so that it can be used to measure the local forward/backward-incompleteness generated by a sound abstract function $f_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$, operating on a set of approximated properties $\mathcal{A}$, with respect to the concrete function $f_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$, operating on a set of properties $\mathcal{C}$ some of which may be undecidable. The target distance function could be defined either on the concrete domain $\mathcal{C}$ in order to calculate the distance between $f_{\mathcal{C}}(\gamma(a))$ and $\gamma(f_{\mathcal{A}}(a))$ for an input $a \in \mathcal{A}$, that is the local forward-(in)completeness, or, e.g. when they enjoy a GC, directly on the abstract domain $\mathcal{A}$ for measuring the distance between $\alpha(f_{\mathcal{C}}(c))$ and $f_{\mathcal{A}}(\alpha(c))$ for $c \in \mathcal{C}$, that is the local backward-(in)completeness (as, e.g., formalized in [11] through the use of quasi-metrics).

In abstract interpretation both $\mathcal{C}$ and $\mathcal{A}$ are often based on a qualitative notion of precision in order to know when an element is more precise respect to another. More generally, given an unordered set D, a basic relation able to accomplish this task is a pre-order relation $\preceq_D \in \wp(D \times D)$ where $x \preceq_D y$ for $x, y \in D$ intuitively means that y *approximates* x [22]. Therefore, we need to define a general notion of distance able to exploit *any* pre-ordered structure. Let us informally analyze each property that we may expect on a distance measuring the local incompleteness, either backward or forward, of abstract interpretations.

When comparing two elements $x, y \in D$ in a pre-ordered set $(D, \preceq_D)$, the distance function $\delta_D(x, y)$ must return a non-negative real value for all $x, y \in D$. We also give δ_D the possibility to return the symbol ∞ meaning an infinite distance between two elements. Thus, the type of a distance function δ_D will be:

$$\delta_D : D \times D \to \mathbb{R}^{\infty}_{\geq 0} \tag{0}$$

If we are calculating the distance between two identical elements, then we expect δ_D to output zero:

$$x = y \;\Rightarrow\; \delta_D(x, y) = 0 \tag{1}$$

However, we do not require the converse implication: we allow $x \neq y$ even if $\delta_D(x, y) = 0$. This gives us the freedom to say that, e.g., the distance between two distinct elements is zero because the distance itself is considering the information represented by x and y *up to some abstraction of interest*. That is, the distance itself can be considered as *another* layer of approximation between the elements of D and, thus, it may output zero even if they are represented differently in D. For example, consider the poset $(\wp(\mathbb{Z}), \subseteq)$ corresponding to the powerset of integers together with the subset inclusion relation (i.e., a partial-order). Given two sets $X, Y \in \wp(\mathbb{Z})$ such that $X \subseteq Y$ (e.g., $X = \{2, 9, 19\}$ and $Y = \{2, 9, 15, 19\}$), we might be interested in a function $\delta_{\wp(\mathbb{Z})}$ that calculates the distance of an approximated representation of both X and Y, for instance, by taking their interval abstraction. In this case, it might happen that X and Y are mapped to the same interval (i.e., the interval $[2, 19]$ for the chosen X

and Y) and therefore $\delta_{\wp(\mathbb{Z})}(X, Y) = 0$ even though $X \neq Y$. As another example, when considering the convex polyhedra domain $(\mathsf{Poly}, \preceq_{\mathsf{Poly}})$ [24] over $\mathbb{R}^n$ we might want that the distance between two polyhedra $p_1, p_2 \in \mathsf{Poly}$ is zero when they represent the same set of vectors in $\mathbb{R}^n$. That is, if $\gamma(p_1) = \gamma(p_2)$ then $\delta_{\mathsf{Poly}}(p_1, p_2) = 0$ even if p_1 and p_2 are represented by different inequalities in Poly, i.e., $p_1 \neq p_2$.

The requirements (0)-(1) define δ_D to be a generalization of a metric: by relaxing the identity of indiscernibles axiom and dropping the symmetry and triangle inequality axioms of metrics, we get a *pre-metric*[3]. Similarly to the relation between pre-orders and other stronger orderings (e.g., partial-orders and equivalence relations), pre-metrics are more general than pseudoquasi-metrics, quasi-metrics and metrics (see Section 2): a pre-metric satisfying the triangle inequality axiom is a pseudoquasi-metric, furthermore if it also satisfies the identity of indiscernibles then it is a quasi-metric, while a symmetric quasi-metric is a metric. Pre-metrics can be considered as one of the weakest forms of distance functions from which we can build on top of pre-ordered sets.

Until now the definition of pre-metric does not consider the pre-order relation between elements of D. Recall that we are interested in computing a distance between the result of a concrete operator $f_{\mathcal{C}}$ working on $(\mathcal{C}, \preceq_{\mathcal{C}})$ and a *sound* abstract operator $f_{\mathcal{A}}$ working on $(\mathcal{A}, \preceq_{\mathcal{A}})$. Therefore, we already know that for any $a \in \mathcal{A}$ the two results $f_{\mathcal{C}}(\gamma(a))$ and $\gamma(f_{\mathcal{A}}(a))$ are comparable according to $\preceq_{\mathcal{C}}$, namely $f_{\mathcal{C}}(\gamma(a)) \preceq_{\mathcal{C}} \gamma(f_{\mathcal{A}}(a))$ thanks to the soundness assumption of $f_{\mathcal{A}}$. This means that our definition of distance should have a meaning when used to calculate distances between elements being part of the same chain, i.e., comparable according to $\preceq_D$, while we do not care about the result of $\delta_D(x, y)$ when $x \not\preceq_D y$. That said, suppose $x, y, z \in D$ are related by $x \preceq_D y \preceq_D z$, i.e., z is an approximation of y and y approximates x. If we ascend the chain from x to y, then we would expect that the remaining distance from y to z to be less than or equal the entire distance from x to z. Similarly, if we descend the chain from z to y then we would expect the remaining distance from x and y to be less than or equal the whole distance from x to z. Formally:

$$x \preceq_D y \preceq_D z \;\Rightarrow\; \delta_D(x, y) \leq \delta_D(x, z) \wedge \delta_D(y, z) \leq \delta_D(x, z) \tag{2}$$

This axiom gives us the possibility to reason on distance results between elements on the same chain. For example, suppose that the concrete and abstract domains are related by a GC $(\mathcal{C}, \preceq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_{\mathcal{A}})$ and that we have defined a distance $\delta_{\mathcal{A}}$ on the elements of $\mathcal{A}$. Given an input $c \in \mathcal{C}$, we already know that the result of the bca of $f_{\mathcal{C}}$ on $\mathcal{A}$ is in the middle between the abstraction of $f_{\mathcal{C}}(c)$ and the result of the abstract sound operator $f_{\mathcal{A}}(\alpha(c))$, namely, it holds $\alpha(f_{\mathcal{C}}(c)) \preceq_{\mathcal{A}} f^{\alpha}_{\mathcal{A}}(\alpha(c)) \preceq_{\mathcal{A}} f_{\mathcal{A}}(\alpha(c))$. In this case, we would expect that the distance between $\alpha(f_{\mathcal{C}}(c))$ and the best possible approximation of $f_{\mathcal{C}}$, i.e., $\delta_{\mathcal{A}}(\alpha(f_{\mathcal{C}}(c)), f^{\alpha}_{\mathcal{A}}(\alpha(c)))$ to be less than or equal to the distance between the

[3] This is not a standard term in the literature: sometimes it is used to refer to other generalizations of metrics such as pseudosemi-metrics [8] or pseudo-metrics [34]; it sometimes appears as pra-metric [3]. This definition is taken from Wikipedia [1].

concrete and the chosen abstract operator $f_{\mathcal{A}}$, namely, $\delta_{\mathcal{A}}(\alpha(f_{\mathcal{C}}(c)), f_{\mathcal{A}}(\alpha(c)))$, and the same for $\delta_{\mathcal{A}}(f^{\alpha}_{\mathcal{A}}(\alpha(c)), f_{\mathcal{A}}(\alpha(c)))$. Note that the triangle inequality axiom required by metrics and some of their weakening, like pseudo-metrics and quasi-metrics, does not imply axiom (2), and (2) does not imply the triangle inequality. For example, if $D = \{x, y, z\}$ with $x \preceq_D y \preceq_D z$ and $\delta_D(x, y) = 2$, $\delta_D(y, z) = 1$, $\delta_D(x, z) = 1$, then $\delta_D(x, z) = 1 < 3 = \delta_D(x, y) + \delta_D(y, z)$ but $\delta_D(x, y) = 2 > \delta_D(x, z) = 1$. Instead, if $\delta_D(x, y) = 1$, $\delta_D(y, z) = 1$, $\delta_D(x, z) = 3$ then (2) holds while $\delta_D(x, z) = 3 > 2 = \delta_D(x, y) + \delta_D(y, z)$. In fact, we do not require the triangle inequality axiom (neither its weaker form on chains as formalized, e.g., in [25,36,11]): as we are focusing on incompleteness results and, therefore, elements on chains according to the ordering $\preceq_D$, the distance $\delta_D(x, z)$ could be greater or lower than the sum between $\delta_D(x, y)$ and $\delta_D(y, z)$ as long as it respects (2).

We now have all the ingredients needed to formalize the distance that matches our purposes: it must be a pre-metric (axioms (0)-(1)) compatible with the underlying pre-order (axiom (2)). Functions that meet these requirements over a pre-ordered set $(D, \preceq_D)$ are called *pre-metrics $\preceq_D$-compatible*.

Definition 6 (Pre-metric $\preceq_D$-compatible). *Let $(D, \preceq_D)$ be a pre-ordered set. The function $\delta_D : D \times D \to \mathbb{R}^{\infty}_{\geq 0}$ is a pre-metric $\preceq_D$-compatible if and only if the following axioms are satisfied for all $x, y, z \in D$:*

(1) $x = y \;\Rightarrow\; \delta_D(x, y) = 0$;

(2) $x \preceq_D y \preceq_D z \;\Rightarrow\; \delta_D(x, y) \leq \delta_D(x, z) \wedge \delta_D(y, z) \leq \delta_D(x, z)$.

■

Pre-ordered sets equipped with a compatible pre-metric are called *pre-metric $\preceq_D$-compatible spaces*.

Definition 7 (Pre-metric $\preceq_D$-compatible space). *Given a pre-ordered set $(D, \preceq_D)$ and a pre-metric $\preceq_D$-compatible δ_D, the triple $(D, \preceq_D, \delta_D)$ is a pre-metric $\preceq_D$-compatible space. We use $Pre((D, \preceq_D))$ to refer to the the set of all pre-metric $\preceq_D$-compatible spaces: $(D, \preceq_D, \delta_D) \in Pre((D, \preceq_D))$.* ■

The following is a list of pre-metrics compatible with a generic pre-ordered set $(D, \preceq_D)$ or tailored for specific domains.

Example 1 (Zero-distance). One of the most trivial pre-metric $\preceq_D$-compatible definable on any pre-ordered set is the distance that always returns the value zero for all $x, y \in D$:

$$\delta^0_D(x, y) \stackrel{def}{=} 0$$

Although it satisfies all axioms from Definition 6, it does not provide any information about the distance between elements in D since it treats them as they are close to each other. ♦

Example 2 (Ordering-distance). The following distance

$$\delta_D^{\preceq_D}(x,y) \stackrel{def}{=} \begin{cases} 0 & \text{if } x = y, \\ 1 & \text{if } x \neq y \wedge x \preceq_D y, \\ \infty & \text{otherwise} \end{cases}$$

is clearly a pre-metric $\preceq_D$-compatible. In fact, it extends the pre-order relation $\preceq_D$ with the function $\delta_D^{\preceq_D}$ having three output values: 0 for equal elements, 1 for not equal but comparable elements and ∞ for non-comparable elements. ♦

Example 3 (Measure-distance). Let (Z, D, μ) be a measure space, i.e., D be a domain that forms a σ-algebra over a set Z and $\mu : D \to \mathbb{R}^{\infty}_{\geq 0}$ be a measure function. We define the function δ_D^{μ} for every $X, Y \in D$ as follows:

$$\delta_D^{\mu}(X,Y) \stackrel{def}{=} Av(\mu(Y) - \mu(X))$$

where Av is the absolute value function. Note that, because D is composed by measurable properties, δ_D^{μ} can exploit the measure function μ in order to quantify the distance between elements of D. However, depending on how $\preceq_D$ is defined, it still may not be a pre-metric $\preceq_D$-compatible as axiom (2) may be violated. Let us show two examples where δ_D^{μ} is compatible with $\preceq_D$.

Consider the measure space $(D, \wp(D), \mu^c)$, where $(\wp(D), \subseteq)$ and μ^c is the *counting measure*, namely, for all $X \in \wp(D)$, $\mu^c(X) \stackrel{def}{=} |X|$ if $|X|$ is finite, ∞ otherwise. Intuitively, $\delta_{\wp(D)}^{\mu^c}(X,Y)$ counts the elements in X and Y and returns the absolute value of their difference. Note that: axioms (0)-(1) are satisfied since $\delta_{\wp(D)}^{\mu^c}(X,Y)$ is either non-negative or ∞, and if $X = Y$ then they have the same number of elements which implies[4] $\delta_{\wp(D)}^{\mu^c}(X,Y) = 0$. Furthermore, axiom (2) holds as $X \subseteq Y \subseteq Z$ implies that Z has more elements than Y and Y has more elements than X, thus ascending (resp. descending) a chain implies that the distance will increase (resp. decrease). The function $\delta_{\wp(D)}^{\mu^c}$ fulfills all axioms (0)-(2) and, therefore, it is a pre-metric $\subseteq$-compatible. Dually, the same reasoning holds with $\wp(D)$ being partially-ordered by $\supseteq$. This is one of the most common distance used for evaluating the outcome of a program analysis: you simply count the elements generated by the abstract analysis and the elements generated by the concrete execution and then the absolute value of the difference tells you the quality of the analysis result. The bigger this difference is, the worse the result will be. ♦

Example 4 (Volume-distance). Let us consider the pre-ordered domain of convex polyhedra $(\mathsf{Poly}, \preceq_{\mathsf{Poly}})$. We define the pre-metric

$$\delta_{\mathsf{Poly}}^{Vol}(p_1,p_2) \stackrel{def}{=} Av(Vol(p_1) - Vol(p_2))$$

[4] We assume the following results when the ∞ symbol is involved: $Av(k - \infty) = Av(\infty - k) = \infty$ with $k \in \mathbb{R}$, while $\infty - \infty = 0$.

calculating the absolute value of the difference between the volume of two convex polyhedra $p_1, p_2 \in \mathsf{Poly}$. The volume function $\mathit{Vol} : \mathsf{Poly} \to \mathbb{R}^{\infty}_{\geq 0}$ could be a monotone (namely, if $\gamma(p_1) \subseteq \gamma(p_2)$ then $\mathit{Vol}(p_1) \leq \mathit{Vol}(p_2)$) overapproximation of the exact volume computation (see, e.g., [15,35]). This means that Vol may not be a measure according to Definition 1 as the countable-additivity axiom may be violated. However, $\delta^{\mathit{Vol}}_{\mathsf{Poly}}$ satisfies the two axioms of Definition 6 and therefore it is $\preceq_{\mathsf{Poly}}$-compatible. ♦

Example 5 (Trace-Length distance). Let Σ be a set of program states and let $\Sigma^{+\infty} \stackrel{def}{=} \Sigma^{+} \cup \Sigma^{\infty}$ be the set of all non-empty finite (Σ^{+}) and infinite (Σ^{∞}) sequences of program states. We consider the domain of sets of program traces ordered by set inclusion, i.e., $(\wp(\Sigma^{+\infty}), \subseteq)$, and define the following function $\mathit{Len} \colon \wp(\Sigma^{+\infty}) \to \mathbb{R}^{\infty}_{\geq 0}$:

$$\mathit{Len}(T) \stackrel{def}{=} \begin{cases} 0 & \text{if } T = \varnothing, \\ \max\{|\sigma| \mid \sigma \in T\} & \text{if } T \cap \Sigma^{\infty} = \varnothing, \\ \infty & \text{otherwise} \end{cases}$$

where $|\sigma|$ applied on a trace denotes its length. Len computes the length of the longest program trace in a set of traces T. The following pre-metric

$$\delta^{\mathit{Len}}_{\wp(\Sigma^{+\infty})}(T_1, T_2) \stackrel{def}{=} \mathit{Av}(\mathit{Len}(T_1) - \mathit{Len}(T_2))$$

looking at the absolute value of the difference between the lengths of the longest traces in two sets $T_1, T_2 \in \wp(\Sigma^{+\infty})$ is a pre-metric $\subseteq$-compatible. Note that $\mathit{Len}(T)$ does not form a measure as the countable-additivity axiom does not hold. ♦

Example 6 (Weighted path-length distance). We consider the weighted path-length distance $\delta^{\mathfrak{w}}_{D}$ defined in [11] for posets. We propose a slightly modified version able to work with any pre-ordered structures $(D, \preceq_D)$. Intuitively, $\delta^{\mathfrak{w}}_{D}$ considers a pre-ordered set as a directed weighted graph where the set of edges $E_D \subseteq D \times D$ is defined as $E_D \stackrel{def}{=} \{(x, y) \mid x \prec_D y\}$, and $\mathfrak{w} : E_D \to \mathbb{R}_{\geq 0}$ is the weight function which assigns a non-negative real value to each edge. The relation $x \prec_D y$ is true whenever $x \prec_D y$ and there is no element $z \in D$ such that $x \prec_D z \prec_D y$. Clearly, if $\preceq_D$ is a partial-order then the graph is acyclic. Given $x, y \in D$ such that $x \neq y$, let $\mathfrak{C}^{y}_{x}$ denotes the set of all possible chains $\mathbf{c} \subseteq E_D$ between x and y such that if $(z, u) \in \mathbf{c}$ then $x \preceq_D z \prec_D u \preceq_D y$. It is clear that if $x \not\preceq_D y$ then $\mathfrak{C}^{y}_{x} = \varnothing$. The weighted path-length distance $\delta^{\mathfrak{w}}_{D} : D \times D \to \mathbb{R}^{\mathfrak{w}}_{\geq 0}$ is defined as follows:

$$\delta^{\mathfrak{w}}_{D}(x, y) \stackrel{def}{=} \begin{cases} 0 & \text{if } x = y, \\ \infty & \text{if } \forall \mathbf{c} \in \mathfrak{C}^{y}_{x}.\ |\mathbf{c}| = \omega, \\ \min\left\{ \sum_{e \in \mathbf{c}} \mathfrak{w}(e) \;\middle|\; \begin{array}{l} \mathbf{c} \in \mathfrak{C}^{y}_{x} \\ |\mathbf{c}| < \omega \end{array} \right\} & \text{if } \exists \mathbf{c} \in \mathfrak{C}^{y}_{x}.\ |\mathbf{c}| < \omega. \end{cases}$$

Intuitively, when $\delta^{\mathfrak{w}}_{D}$ is used to calculate the distance between x and y such that $x \prec_D y$ then it outputs the minimum weighted path w.r.t. $\mathfrak{w}$ between x and y,

while if $x \not\preceq_D y$ then it outputs ∞. Note that $\delta_D^{\mathfrak{w}}$ is a pre-metric that does not satisfy symmetry, while it satisfies the triangle inequality axiom only on chains. However, it may not be compatible with the underlying ordering $\preceq_D$ as axiom (2) may turn false. For instance, consider the $\mathsf{Sign} \stackrel{def}{=} \{\mathbb{Z}, -, 0, +, \varnothing\}$ domain for sign analysis of integer variables [19]. Sign is ordered by the following partial-order: $\varnothing \preceq_{\mathsf{Sign}} 0 \preceq_{\mathsf{Sign}} - \preceq_{\mathsf{Sign}} \mathbb{Z}$ and $\varnothing \preceq_{\mathsf{Sign}} 0 \preceq_{\mathsf{Sign}} + \preceq_{\mathsf{Sign}} \mathbb{Z}$. Suppose the weight function $\mathfrak{w}$ assigns $\mathfrak{w}((0,-)) = 5$ while for the others couple $(a,b) \in E_{\mathsf{Sign}}$, $\mathfrak{w}((a,b)) = 1$. Then, the weighted path-length $\delta_{\mathsf{Sign}}^{\mathfrak{w}}$ is not a pre-metric $\preceq_{\mathsf{Sign}}$-compatible as $\delta_{\mathsf{Sign}}^{\mathfrak{w}}(0,-) = 5 > 2 = \delta_{\mathsf{Sign}}^{\mathfrak{w}}(0,\mathbb{Z})$ thus violating (2). On the other hand, if we set $\forall (a,b) \in E_{\mathsf{Sign}}$, $\mathfrak{w}((a,b)) = 1$ then we get a pre-metric $\preceq_{\mathsf{Sign}}$-compatible.

As a final case of application of $\delta_D^{\mathfrak{w}}$, consider the domain of integer intervals Int also known as the box domain. Given any two intervals $i_1, i_2 \in \mathsf{Int}$ such that $(i_1, i_2) \in E_{\mathsf{Int}}$, if we define $\mathfrak{w}((i_1, i_2)) = 1$, then $\delta_{\mathsf{Int}}^{\mathfrak{w}}$ is a pre-metric $\preceq_{\mathsf{Int}}$-compatible. Intuitively, $\delta_{\mathsf{Int}}^{\mathfrak{w}}(i_1, i_2)$ for $i_1 \preceq_{\mathsf{Int}} i_2$ counts how many more elements one interval has compared to the other: if $\delta_{\mathsf{Int}}^{\mathfrak{w}}(i_1, i_2) = k$ for some $k \in \mathbb{N}$, then the interval i_2 contains exactly k more elements than i_1. For instance, $\delta_{\mathsf{Int}}^{\mathfrak{w}}([0,0],[-1,2]) = 3$ as the interval $[-1,2]$ has 3 more elements than the singleton $[0,0]$, namely: $-1, 1, 2$; $\delta_{\mathsf{Int}}^{\mathfrak{w}}([0,10],[0,+\infty]) = \infty$ as $[0,+\infty]$ has an infinite number of more elements than $[0,10]$, while $\delta_{\mathsf{Int}}^{\mathfrak{w}}([0,+\infty],[-5,+\infty]) = 5$. ♦

When a pre-metric $\preceq_D$-compatible is precise enough to assign zero only when two comparable elements are identical, namely, when it satisfies the identity of indiscernibles axiom on chains, it will be called *strong*.

Definition 8 (Strong pre-metric $\preceq_D$-compatible). *Consider the pre-ordered space $(D, \preceq_D, \delta_D)$. The pre-metric $\preceq_D$-compatible δ_D is said to be strong if and only if the following implication holds for every $x, y \in D$:*

$$x \preceq_D y \;\Rightarrow\; (\delta_D(x,y) = 0 \;\Rightarrow\; x = y) \qquad \blacksquare$$

For instance, the ordering-distance $\delta_D^{\preceq_D}$ of Example 2, the weighted path-length $\delta_{\mathsf{Int}}^{\mathfrak{w}}$ defined on intervals in Example 6, $\delta_{\mathsf{Poly}}^{Vol}$ of Example 4 with *Vol* calculating the exact volume, and the counting measure-distance on integer sets $\delta_{\wp(\mathbb{Z})}^{\mu^c}$, are strong. Conversely, the zero-distance δ_D^0 of Examples 1, the volume-distance $\delta_{\mathsf{Poly}}^{Vol}$ with *Vol* overapproximating the real volume, and the trace-length distance $\delta_{\wp(\Sigma^{+\infty})}^{Len}$ defined in Example 5, are not. We will see in Section 5 that strong pre-metrics $\preceq_D$-compatible play an important rule when measuring the local forward/backward-incompleteness of abstract interpretations.

As a last note, it is worth noting that Definition 6 is general enough to be instantiated with other definitions of metrics specifically tailored in the context of abstract interpretation. For instance, if a pre-metric $\preceq_D$-compatible δ_D is also symmetric and it satisfies the weak triangle inequality then it is a pseudo-metric $\preceq_D$-compatible according to [36], whereas if δ_D both induces the underlying order relation, it is strong and it satisfies the weak triangle inequality then it is a quasi-metric $\preceq_D$-compatible [11,25].

4 Deriving Pre-Metrics from Domains

Concrete $\mathcal{C}$ and abstract $\mathcal{A}$ domains of properties in abstract interpretation are often related by a monotonic concretization function $\gamma : \mathcal{A} \to \mathcal{C}$ and sometimes additionally by a monotonic abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$ that maps a concrete element to the best (i.e., the smallest according to $\preceq_{\mathcal{A}}$) abstract element approximating it, such that $(\mathcal{C}, \preceq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_{\mathcal{A}})$ forms a GC. By exploiting these structures we can *derive* pre-metrics from one domain to another.

Given a pre-metric compatible with the concrete domain $(\mathcal{C}, \preceq_{\mathcal{C}})$, we can exploit the concretization function γ to derive a pre-metric compatible with the underlying abstract domain ordering $(\mathcal{A}, \preceq_{\mathcal{A}})$. Here a GC between $\mathcal{C}$ and $\mathcal{A}$ is not necessary.

Definition 9 (Induced distance from the concrete domain). *Consider* $(\mathcal{C}, \preceq_{\mathcal{C}}, \delta_{\mathcal{C}}) \in Pre((\mathcal{C}, \preceq_{\mathcal{C}}))$. *For all* $a_1, a_2 \in \mathcal{A}$, *we define:*

$$\overrightarrow{\delta}_{\mathcal{A}}(a_1, a_2) \overset{def}{=} \delta_{\mathcal{C}}(\gamma(a_1), \gamma(a_2))$$

as the pre-metric induced on $\mathcal{A}$ *from* $(\mathcal{C}, \preceq_{\mathcal{C}}, \delta_{\mathcal{C}})$. ■

Proposition 1. *The following statements hold:*

(i) $(\mathcal{A}, \preceq_{\mathcal{A}}, \overrightarrow{\delta}_{\mathcal{A}})$ *is a pre-metric* $\preceq_{\mathcal{A}}$*-compatible space;*

(ii) if $\delta_{\mathcal{C}}$ *is strong and* γ *is injective then* $\overrightarrow{\delta}_{\mathcal{A}}$ *is strong.* □

Furthermore, given $(\mathcal{A}, \preceq_{\mathcal{A}}, \delta_{\mathcal{A}}) \in Pre((\mathcal{A}, \preceq_{\mathcal{A}}))$, when the concrete and abstract domains admit a GC through an abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$, we can derive the pre-metric $\preceq_{\mathcal{C}}$-compatible on the concrete properties $(\mathcal{C}, \preceq_{\mathcal{C}})$. This distance will be called the induced distance from the abstract pre-metric $\preceq_{\mathcal{A}}$-compatible space.

Definition 10 (Induced distance from the abstract domain). *Let the concrete and abstract domains be correlated by a GC* $(\mathcal{C}, \preceq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_{\mathcal{A}})$. *Moreover, let* $(\mathcal{A}, \preceq_{\mathcal{A}}, \delta_{\mathcal{A}}) \in Pre((\mathcal{A}, \preceq_{\mathcal{A}}))$ *be a pre-metric* $\preceq_{\mathcal{A}}$*-compatible space. For all* $c_1, c_2 \in \mathcal{C}$, *we define:*

$$\overleftarrow{\delta}_{\mathcal{C}}(c_1, c_2) \overset{def}{=} \delta_{\mathcal{A}}(\alpha(c_1), \alpha(c_2))$$

as the pre-metric induced on $\mathcal{C}$ *from* $(\mathcal{A}, \preceq_{\mathcal{A}}, \delta_{\mathcal{A}})$. ■

Proposition 2. $(\mathcal{C}, \preceq_{\mathcal{C}}, \overleftarrow{\delta}_{\mathcal{C}})$ *is a pre-metric* $\preceq_{\mathcal{C}}$*-compatible space.* □

The derived pre-metric on the concrete properties is compatible with $\preceq_{\mathcal{C}}$ as it measures the distance between two concrete elements by throwing away non-relevant information according to the abstraction α.

Note how Definition 9 and Definition 10 define a way to build pre-metrics on domains correlated by a concretization function and/or an abstraction function.

This means that the distance itself δ_D defined on a pre-ordered domain D, can view properties of D on different levels of precision: δ_D can exploit a more approximated pre-metric $\delta_{\mathcal{A}}$ defined on an abstraction of properties $\mathcal{A}$ of D, e.g. we can use $\overleftarrow{\delta}_D$ when $(D, \preceq_D) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_{\mathcal{A}})$. Alternatively, δ_D can exploit a more precise distance $\delta_{\mathcal{C}}$, for instance when $\delta_{\mathcal{C}}$ is defined on a more precise domain $\mathcal{C}$ related with D through the concretization $\gamma : D \to \mathcal{C}$, then we can use $\overrightarrow{\delta}_D$. We can also combine distances in a way similar to combining abstractions.

Example 7. Let Zone be the zone domain [37], and Oct be the octagon domain [38]. Both are relational domains, with Oct more precise than Zone, able to infer affine relationships (inequalities) between variables, although in a more restricted form respect to Poly. Consider the volume-distance $\delta^{Vol}_{\mathsf{Poly}}$ defined on convex polyhedra in Example 4. We can systematically derive other volume-distances on domains which can be represented by Poly, e.g., Int, Zone and Oct. For instance, given $\gamma_{\mathsf{Oct}} : \mathsf{Oct} \to \mathsf{Poly}$, $\gamma_{\mathsf{Zone}} : \mathsf{Zone} \to \mathsf{Poly}$, $\gamma_{\mathsf{Int}} : \mathsf{Int} \to \mathsf{Poly}$, for all $o_1, o_2 \in \mathsf{Oct}$, $z_1, z_2 \in \mathsf{Zone}$, $i_1, i_2 \in \mathsf{Int}$ we get

$$\overrightarrow{\delta}^{Vol}_{\mathsf{Oct}}(o_1, o_2) = \delta^{Vol}_{\mathsf{Poly}}(\gamma_{\mathsf{Oct}}(o_1), \gamma_{\mathsf{Oct}}(o_2))$$
$$\overrightarrow{\delta}^{Vol}_{\mathsf{Zone}}(z_1, z_2) = \delta^{Vol}_{\mathsf{Poly}}(\gamma_{\mathsf{Zone}}(z_1), \gamma_{\mathsf{Zone}}(z_2))$$
$$\overrightarrow{\delta}^{Vol}_{\mathsf{Int}}(i_1, i_2) = \delta^{Vol}_{\mathsf{Poly}}(\gamma_{\mathsf{Int}}(i_1), \gamma_{\mathsf{Int}}(i_2)) \qquad \blacklozenge$$

Depending on a number of factors such as the imprecision we want to track, the quantity of information represented by a domain, and/or the computational complexity needed to implement δ_D, we may switch from one domain to another. This procedure is also common in program analysis by abstract interpretation where it can be useful to convert between one abstract domain and another, for instance to switch abstract domains dynamically during the analysis or benefit from abstract operators available in other more abstract domains (see, e.g., [18,39]).

Example 8. Let us consider as concrete domain $(\wp(\mathbb{Z}^n), \subseteq)$ and the abstract pre-metric $\preceq_{\mathsf{Int}}$-compatible space $(\mathsf{Int}, \preceq_{\mathsf{Int}}, \delta^{\mathfrak{w}}_{\mathsf{Int}})$ of intervals together with the weighted path-length defined in Example 6. We can derive the pre-metric

$$\overleftarrow{\delta}_{\wp(\mathbb{Z}^n)}(S_1, S_2) = \delta^{\mathfrak{w}}_{\mathsf{Int}}(\alpha_i(S_1), \alpha_i(S_2))$$

where for all $S_1, S_2 \in \wp(\mathbb{Z}^n)$, $\alpha_i : \wp(\mathbb{Z}^n) \to \mathsf{Int}$ calculates the interval of the i-th component only, with $1 \leq i \leq n$, of set of vectors S_1 and S_2. For instance, if $n = 3$ and $S_1 = \{\langle 1, 9, 9\rangle, \langle 1, 0, 10\rangle\}$, $S_2 = \{\langle 1, 5, 0\rangle, \langle -1, 0, 10\rangle, \langle 5, 0, 0\rangle\}$ then $\alpha_1(S_1) = [1, 1]$, $\alpha_1(S_2) = [-1, 5]$, and their distance is $\overleftarrow{\delta}_{\wp(\mathbb{Z}^n)}(S_1, S_2) = \delta^{\mathfrak{w}}_{\mathsf{Int}}([1, 1], [-1, 5]) = 6$. This can be useful, e.g., when $\sigma \in S$ represents a program state and the i-th component of σ corresponds to the value of a program variable, thus, $\overleftarrow{\delta}_{\wp(\mathbb{Z}^n)}(S_1, S_2)$ is interested in calculating the imprecision of that variable only. $\blacklozenge$

5 Partial Forward/Backward-Completeness Properties

As already mentioned in Section 2, Campion et al. in [11] proposed a relaxation of the local backward-completeness through the use of quasi-metrics, leading to Definition 5. More specifically, they require $\mathcal{C}$ and $\mathcal{A}$ to be related by a GC. In their formalization, the use of quasi-metrics enforces distance functions to adhere precisely to the underlying partial-ordering (namely, returning a distance for comparable elements only), and to output zero only when both elements are equal (corresponding to (1) but with both implications). These conditions imply that, if we want to define a quasi-metric on the concrete properties $\mathcal{C}$, the distance function must be precise enough to distinguish when two concrete elements are equal, thus limiting the possibility to choose, e.g., computationally less expensive distances at the cost of losing precision. For instance, the volume-distance defined in Example 4 on Poly would not be possible as a quasi-metric Poly-compatible unless *Vol* exactly calculates the volume of a convex polytope, which has exponential complexity. Similarly, defining a distance that partially considers the information encoded in the concrete elements, e.g. the imprecision of a specific program variable (Example 8), is not allowed.

In this section we exploit the newly introduced notion of pre-metrics $\preceq_D$-compatible to weaken both definitions of local forward-completeness (Definition 3) and local backward-completeness (Definition 4). Thanks to the weaker requirements of pre-metrics, we ask $\mathcal{C}$ and $\mathcal{A}$ to have fewer structures compared to [11]: they must be, at least, pre-ordered sets and be correlated by a monotone concretization function $\gamma : \mathcal{A} \to \mathcal{C}$ but not necessary forming a GC with an abstraction function α. Weakening the local forward-completeness property involves defining a pre-metric $\preceq_{\mathcal{C}}$-compatible on the concrete domain $(\mathcal{C}, \preceq_{\mathcal{C}})$: this can be achieved by either defining a pre-metric specifically tailored for $(\mathcal{C}, \preceq_{\mathcal{C}})$ or, as showed in Section 4, by deriving a distance from another domain which may approximate the computation. The new notion of ε-partial forward-completeness is defined as follows.

Definition 11 (ε-Partial forward-completeness). *Let us consider a pre-metric $\preceq_{\mathcal{C}}$-compatible space $(\mathcal{C}, \preceq_{\mathcal{C}}, \delta_{\mathcal{C}}) \in Pre((\mathcal{C}, \preceq_{\mathcal{C}}))$ and let $f_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ be a sound approximation of $f_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$. Given $\varepsilon \in \mathbb{R}^{\infty}_{\geq 0}$, we say that $f_{\mathcal{A}}$ is an ε-partial forward-complete approximation of $f_{\mathcal{C}}$ on input $a \in \mathcal{A}$ if and only if the following predicate holds:*

$$\delta_{\mathcal{C}}(f_{\mathcal{C}}(\gamma(a)), \gamma(f_{\mathcal{A}}(a))) \leq \varepsilon \qquad \blacksquare$$

The value of the distance $\delta_{\mathcal{C}}$ between the result of the concrete operator $f_{\mathcal{C}}(\gamma(a))$ and the concretization of the abstract operator $\gamma(f_{\mathcal{A}}(a)))$ can be interpreted as the measure of the approximation introduced by $f_{\mathcal{A}}$ with respect to $f_{\mathcal{C}}$ at input a. Therefore, this distance encodes a quantitative level of imprecision introduced by $f_{\mathcal{A}}$, more precisely, *the imprecision that we want to measure* according to how we have defined the pre-metric $\preceq_{\mathcal{C}}$-compatible $\delta_{\mathcal{C}}$.

```
var x : int, y : int;
begin
  x = 0; y = 0;
  while (x <= 9) and (y >= 0) do
    if x <= 4 then
      x = x + 1; y = y + 1;
    else
      x = x + 1; y = y − 1;
    endif;
  done;
end
```

Fig. 1: The Program P

```
var x : int;
begin
  while x > 0 do
    x = x − 1;
  done;
end
```

Fig. 2: The Program Q

Example 9 (Static analysis of numeric invariants). We want to analyze the partial forward-completeness of the Interproc[5] [2] static analyzer when used to infer the numerical invariant of the while-loop of program P defined in Fig. 1 using the abstract domains $\mathcal{A} \in \{\mathsf{Oct}, \mathsf{Poly}\}$. The imprecision generated by the abstract execution $[\![P]\!]_{\mathcal{A}}$ with respect to the concrete (collecting) execution $[\![P]\!]$, is measured by using the following pre-metric $\subseteq$-compatible on $(\wp(\mathbb{Z}^n), \subseteq)$:

$$\%Vol(S_1, S_2) \stackrel{def}{=} \frac{(Vol(\alpha_{\mathsf{Int}^n}(S_2)) - Vol(\alpha_{\mathsf{Int}^n}(S_1))) \cdot 100}{Vol(\alpha_{\mathsf{Int}^n}(S_1))}$$

Intuitively, the value returned by $\%Vol(S_1, S_2)$ is to be interpreted as the percentage of more volume that the abstraction $(\alpha_{\mathsf{Int}^n})$ of S_2 into Int^n has compared to the volume of the abstraction of S_1 into Int^n, namely, $Vol(\alpha_{\mathsf{Int}^n}(S_1))$ and $Vol(\alpha_{\mathsf{Int}^n}(S_2))$ are the volumes of the two smallest hyperrectangles containing, respectively, S_1 and S_2. Calculating the exact volume of hyperrectangles is generally much less computationally expensive than computing volumes of octagons and polyhedra, so this choice can be a good trade-off. In our case example, $n = 2$ since P has two variables so that Int^2 represents rectangles and $Vol(\alpha_{\mathsf{Int}^2}(S))$ is the area of the rectangle $\alpha_{\mathsf{Int}^2}(S)$. Note that, since the concrete $[\![P]\!]$ and the two abstract executions $[\![P]\!]_{\mathsf{Poly}}$, $[\![P]\!]_{\mathsf{Oct}}$ respect $[\![P]\!] \subseteq \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Poly}}) \subseteq \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Oct}})$ where $\gamma_{\wp(\mathbb{Z}^2)}$ is the concretization of Oct and Poly into $\wp(\mathbb{Z}^2)$, then, thanks to axiom 2, we are sure that

$$\begin{aligned} \%Vol([\![P]\!], \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Poly}})) &\leq \%Vol([\![P]\!], \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Oct}})) \\ \%Vol(\gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Poly}}), \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Oct}})) &\leq \%Vol([\![P]\!], \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Oct}})) \end{aligned}$$

[5] Interproc is freely available at http://pop-art.inrialpes.fr/interproc/interprocweb.cgi

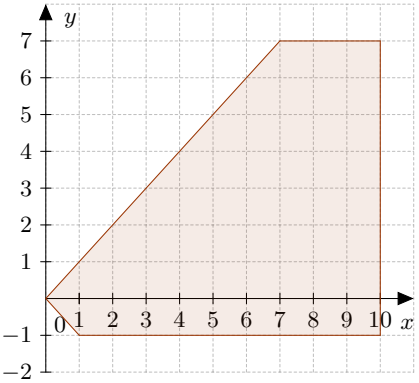


Fig. 3: Loop invariant generated by $[\![P]\!]_{\mathsf{Oct}}$

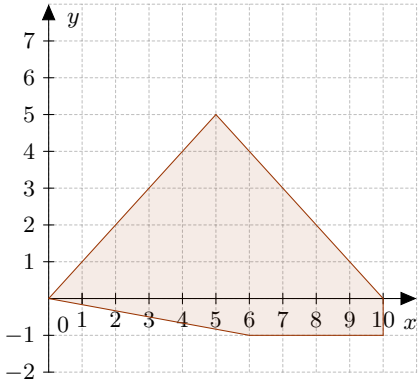


Fig. 4: Loop invariant generated by $[\![P]\!]_{\mathsf{Poly}}$

hold for program P. This means that $\%Vol$ *estimates how more inaccurate* is $[\![P]\!]_{\mathsf{Oct}}$ compared to $[\![P]\!]_{\mathsf{Poly}}$, $[\![P]\!]_{\mathsf{Poly}}$ compared to $[\![P]\!]$, and $[\![P]\!]_{\mathsf{Oct}}$ compared to $[\![P]\!]$.

Suppose our imprecision tolerance measured by $\%Vol$ is 20%. We want to verify when $\%Vol([\![P]\!], \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathcal{A}})) \leq 20$ holds, i.e., whether $[\![P]\!]_{\mathsf{Oct}}$ and $[\![P]\!]_{\mathsf{Poly}}$ are 20-partial forward-complete. By running Interproc using Oct and Poly[6] we get the following inequalities representing the inferred while-loop invariants:

$$\begin{aligned}[\![P]\!]_{\mathsf{Oct}} = \{&x >= 0; -x + 10 >= 0; -x + y + 11 >= 0; x + y >= 0;\\ &y + 1 >= 0; -x - y + 17 >= 0; x - y >= 0; -y + 7 >= 0\}\\ [\![P]\!]_{\mathsf{Poly}} = \{&-x - y + 10 >= 0; -x + 10 >= 0; y + 1 >= 0;\\ &x - y >= 0; x + 6y >= 0\}\end{aligned}$$

Fig. 3 and Fig. 4 depict, respectively, $[\![P]\!]_{\mathsf{Oct}}$ and $[\![P]\!]_{\mathsf{Poly}}$. The pre-metric $\%Vol$ outputs:

$$\begin{aligned}\%Vol(\gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Poly}}), \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Oct}})) &= 33.33\\ \%Vol([\![P]\!], \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Poly}})) &= 20\\ \%Vol([\![P]\!], \gamma_{\wp(\mathbb{Z}^2)}([\![P]\!]_{\mathsf{Oct}})) &= 60\end{aligned}$$

These numbers validate the better accuracy of $[\![P]\!]_{\mathsf{Poly}}$ compared to $[\![P]\!]_{\mathsf{Oct}}$ by providing us a quantitative estimation: the rectangle representing $[\![P]\!]_{\mathsf{Oct}}$ has 33.33% more volume than $[\![P]\!]_{\mathsf{Poly}}$, the one representing $[\![P]\!]_{\mathsf{Poly}}$ has 20% more volume than the concrete execution $[\![P]\!]$, while $[\![P]\!]_{\mathsf{Oct}}$ has 60% more volume than $[\![P]\!]$. We can conclude that $[\![P]\!]_{\mathsf{Poly}}$ is 20-partial forward-complete whereas $[\![P]\!]_{\mathsf{Oct}}$ is not.

[6] For the convex polyhedra analysis, we activated the option of 2 descending steps.

It is worth noting that, the same results can be drawn by defining a similar (computationally more efficient) pre-metric compatible with the Oct domain $\%Vol(o_1, o_2)$ with $o_1, o_2 \in \mathsf{Oct}$ which abstracts octagons into boxes, thus calculating for instance $\%Vol(\alpha_{\mathsf{Oct}}(\llbracket P \rrbracket), \llbracket P \rrbracket_{\mathsf{Oct}})$ without passing through the concrete domain $\wp(\mathbb{Z}^2)$. ♦

When $\mathcal{C}$ and $\mathcal{A}$ enjoy a GC and a pre-metric $\preceq_{\mathcal{A}}$-compatible is defined over the abstract elements of $\mathcal{A}$, then we can weaken the notion of local backward-completeness to obtain the ε-partial backward-completeness property.

Definition 12 (ε-Partial backward-completeness). *Let the concrete and abstract domains be correlated by a GC* $(\mathcal{C}, \preceq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_{\mathcal{A}})$. *Furthermore, assume* $(\mathcal{A}, \preceq_{\mathcal{A}}, \delta_{\mathcal{A}}) \in Pre((\mathcal{A}, \preceq_{\mathcal{A}}))$ *and let* $f_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ *be a sound approximation of* $f_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$. *Given* $\varepsilon \in \mathbb{R}^{\infty}_{\geq 0}$, *we say that* $f_{\mathcal{A}}$ *is an* ε*-partial backward-complete approximation of* $f_{\mathcal{C}}$ *on input* $c \in \mathcal{C}$ *if and only if the following predicate holds:*

$$\delta_{\mathcal{A}}(\alpha(f_{\mathcal{C}}(c)), f_{\mathcal{A}}(\alpha(c))) \leq \varepsilon \qquad \blacksquare$$

The ε-partial backward-completeness property of an abstract sound operator $f_{\mathcal{A}}$ encodes a limited amount of imprecision measured by $\delta_{\mathcal{A}}$, namely at maximum ε, between the abstraction of the concrete execution $\alpha(f_{\mathcal{C}}(c))$ and the abstract execution $f_{\mathcal{A}}(\alpha(c))$ over the concrete input c. Note the difference between the above definition and Definition 5 presented in [11]: here $\mathcal{C}$ and $\mathcal{A}$ are required to be at least pre-orders, and pre-metrics $\preceq_{\mathcal{A}}$-compatible are employed as distance functions, instead of quasi-metrics $\mathcal{A}$-compatible.

We conclude this section by showing some common characteristics between partial forward- and partial backward-completeness properties.

Proposition 3. *Let* $(\mathcal{C}, \preceq_{\mathcal{C}}, \delta_{\mathcal{C}}) \in Pre((\mathcal{C}, \preceq_{\mathcal{C}}))$ *and* $f_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ *be a correct approximation of* $f_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$. *The following hold for every* $a \in \mathcal{A}$*:*

(i) $f_{\mathcal{A}}$ *ε-partial forward-complete at* $a \;\Rightarrow\; \forall \xi \geq \varepsilon : f_{\mathcal{A}}$ *ξ-partial forward-complete at* a*;*

(ii) $f_{\mathcal{A}}$ *∞-partial forward-complete at* a*.* □

Proposition 4. *Let* $(\mathcal{C}, \preceq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_{\mathcal{A}})$ *and* $(\mathcal{A}, \preceq_{\mathcal{A}}, \delta_{\mathcal{A}}) \in Pre((\mathcal{A}, \preceq_{\mathcal{A}}))$. *The following hold for every* $c \in \mathcal{C}$*:*

(i) $f_{\mathcal{A}}$ *ε-partial backward-complete at* $c \;\Rightarrow\; \forall \xi \geq \varepsilon : f_{\mathcal{A}}$ *ξ-partial backward-complete at* c*;*

(ii) $f_{\mathcal{A}}$ *∞-partial backward-complete at* c*.* □

If $f_{\mathcal{A}}$ is ε-partial forward-complete at a (resp. ε-partial backward-complete at c) then admitting a larger imprecision ξ according to $\delta_{\mathcal{C}}$ (resp. $\delta_{\mathcal{A}}$) results in the property of ξ-partial forward-completeness (resp. ξ-partial backward-completeness) which is always satisfied by $f_{\mathcal{A}}$. This implies that, if we define

the class of all ε-partial forward-complete (sound) abstract operators with respect to $f_{\mathcal{C}}$ and input $a \in \mathcal{A}$, and the class of all ε-partial backward-complete (sound) abstract operators with respect to $f_{\mathcal{C}}$ and input $c \in \mathcal{C}$, namely

$$\mathbb{F}^{\varepsilon}_{f_{\mathcal{C}},a} \stackrel{def}{=} \{f_{\mathcal{A}} \mid \delta_{\mathcal{C}}(f_{\mathcal{C}}(\gamma(a)), \gamma(f_{\mathcal{A}}(a))) \leq \varepsilon\}$$
$$\mathbb{B}^{\varepsilon}_{f_{\mathcal{C}},c} \stackrel{def}{=} \{f_{\mathcal{A}} \mid \delta_{\mathcal{A}}(\alpha(f_{\mathcal{C}}(c)), f_{\mathcal{A}}(\alpha(c))) \leq \varepsilon\}$$

then for all $\xi \geq \varepsilon$: $\mathbb{F}^{\varepsilon}_{f_{\mathcal{C}},a} \subseteq \mathbb{F}^{\xi}_{f_{\mathcal{C}},a}$ and $\mathbb{B}^{\varepsilon}_{f_{\mathcal{C}},c} \subseteq \mathbb{B}^{\xi}_{f_{\mathcal{C}},c}$. The second point of Proposition 3 and Proposition 4 simply states that any sound approximation $f_{\mathcal{A}}$ of $f_{\mathcal{C}}$ is partial forward/backward-complete when we admit an infinite level of imprecision.

6 Characterizing Local Forward/Backward-Completeness

In the original definition of partial (backward-)completeness given in [11] using quasi-metrics, asking for 0-partial (backward-)completeness at an input $c \in \mathcal{C}$ is equivalent to require local backward-completeness at c [11].

In our more relaxed framework where pre-metrics are involved and the identity of indiscernibles axiom is not satisfied, requiring 0-partial backward-completeness at c may not be the same as demanding local backward-completeness at c and, similarly, requiring 0-partial forward-completeness at input $a \in \mathcal{A}$ may not coincide with local forward-completeness at a. This is a consequence of the possible approximation introduced by the pre-metric when valuating the distance.

Example 10. Given $(\wp(\mathbb{Z}^2), \subseteq)$, consider the pre-metric $\subseteq$-compatible $\%Vol$ defined in Example 9, and the two sets $S_1 = \{\langle 0,2\rangle, \langle 3,5\rangle\}$, $S_2 = \{\langle 0,2\rangle, \langle 1,1\rangle, \langle 3,5\rangle\}$ such that $S_1 \subseteq S_2$. The $\%Vol$ between S_1 and S_2 is $\%Vol(S_1, S_2) = 0$ even if $S_1 \neq S_2$. This is because of the approximation made by $\%Vol$ which considers an approximated representation of S_1 and S_2, namely, rectangles so that $\alpha_{\mathsf{Int}^2}(S_1) = \langle [0,3], [1,5]\rangle = \alpha_{\mathsf{Int}^2}(S_2)$. Therefore, if S_1 and S_2 are the results of, respectively, a concrete operator $f_{\wp(\mathbb{Z}^2)}$ and and abstract operator f_{Int^2}, then f_{Int^2} is 0-partial forward-complete but not local forward-complete. ♦

However, it turns out that the 0-partial forward-completeness property coincides with the local forward-completeness property when the pre-metric $\preceq_{\mathcal{C}}$-compatible is strong.

Theorem 1. *If $\delta_{\mathcal{C}}$ is strong then the following equivalence holds for all $a \in \mathcal{A}$:*

$$f_{\mathcal{A}} \text{ 0-partial forward-complete at } a \Leftrightarrow f_{\mathcal{A}} \text{ locally forward-complete at } a \quad \square$$

Example 11. If we define the weighted path-length directly on $(\wp(\mathbb{Z}), \subseteq)$, namely, $\delta^{\mathfrak{w}}_{\wp(\mathbb{Z})}$ where $\mathfrak{w}(S_1, S_2) = 1$ for all $(S_1, S_2) \in E_{\wp(\mathbb{Z})}$, then $\delta^{\mathfrak{w}}_{\wp(\mathbb{Z})}$ is strong. Consider the program Q defined in Fig. 2. We analyze the value of variable x at the end of the program having input the interval $[10, 10]$ using Interproc on the interval abstract domain Int with no widening at the first 10 loop iterations. The result

of the analysis is $\gamma([\![Q]\!]_{\mathsf{Int}}[10,10]) = \{0\} = [\![Q]\!]\gamma([10,10])$, and the weighted path-length outputs

$$\delta^{\mathfrak{w}}_{\wp(\mathbb{Z})}([\![Q]\!]\gamma([10,10]), \gamma([\![Q]\!]_{\mathsf{Int}}[10,10])) = 0$$

i.e., $[\![Q]\!]_{\mathsf{Int}}$ is 0-partial forward-complete on input $[10,10]$ using $\delta^{\mathfrak{w}}_{\wp(\mathbb{Z})}$. Since $\delta^{\mathfrak{w}}_{\wp(\mathbb{Z})}$ is strong, then we are sure that $[\![Q]\!]\gamma([10,10]) = \gamma([\![Q]\!]_{\mathsf{Int}}[10,10])$, i.e., $[\![Q]\!]_{\mathsf{Int}}$ is locally forward-complete. ♦

A similar reasoning also applies to the 0-partial backward-completeness property when strong pre-metrics $\preceq_{\mathcal{A}}$-compatible are employed.

Theorem 2. *If $\delta_{\mathcal{A}}$ is strong then the following equivalence holds for all $c \in \mathcal{C}$:*

$$f_{\mathcal{A}} \text{ 0-partial backward-complete at } c \Leftrightarrow f_{\mathcal{A}} \text{ locally backward-complete at } c \quad \square$$

As a final observation, in cases where the concrete and abstract domains enjoy a GI through an abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$ and $\mathcal{A}$ is equipped with a strong pre-metric $\preceq_{\mathcal{A}}$-compatible $\delta_{\mathcal{A}}$, we can characterize the local backward-completeness property over exactly representable elements of $\mathcal{C}$ as an instance of the 0-partial forward-completeness property by exploiting the induced pre-metric $\overleftarrow{\delta_{\mathcal{C}}}$ from the abstract pre-metric $\preceq_{\mathcal{A}}$-compatible space.

Theorem 3. *Let $(\mathcal{C}, \preceq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \preceq_{\mathcal{A}})$, $(\mathcal{A}, \preceq_{\mathcal{A}}, \delta_{\mathcal{A}}) \in Pre((\mathcal{A}, \preceq_{\mathcal{A}}))$ and $(\mathcal{C}, \preceq_{\mathcal{C}}, \overleftarrow{\delta_{\mathcal{C}}}) \in Pre((\mathcal{C}, \preceq_{\mathcal{C}}))$. If $\delta_{\mathcal{A}}$ is strong then the following equivalence holds for every $a \in \mathcal{A}$:*

$$f_{\mathcal{A}} \text{ 0-partial forward-complete at } a \Leftrightarrow f_{\mathcal{A}} \text{ locally backward-complete at } \gamma(a) \quad \square$$

Example 12. Consider again Example 11 analyzing program Q defined in Fig. 2. Let us use this time the weighted path-length $\delta^{\mathfrak{w}}_{\mathsf{Int}}$ on intervals for reasoning on the partial backward-completeness at input $\gamma([10,10]) = \{10\}$ of variable x at the end of program Q. Recall that $\delta^{\mathfrak{w}}_{\mathsf{Int}}$ is a strong pre-metric $\preceq_{\mathsf{Int}}$-compatible. We get the following equalities:

$$\overleftarrow{\delta}_{\wp(\mathbb{Z})}([\![Q]\!]\gamma([10,10]), \gamma([\![Q]\!]_{\mathsf{Int}}[10,10])) = \delta^{\mathfrak{w}}_{\mathsf{Int}}(\alpha([\![Q]\!]\gamma([10,10])), [\![Q]\!]_{\mathsf{Int}}[10,10]) = 0$$

namely, $[\![Q]\!]_{\mathsf{Int}}$ is 0-partial forward-complete on input $[10,10]$ using $\overleftarrow{\delta}_{\wp(\mathbb{Z})}$. Since $\delta^{\mathfrak{w}}_{\mathsf{Int}}$ is strong, this implies that $\alpha([\![Q]\!]\gamma([10,10])) = [\![Q]\!]_{\mathsf{Int}}[10,10]$, i.e., $[\![Q]\!]_{\mathsf{Int}}$ is locally backward-complete at $\{10\}$. ♦

7 Related Work

Forward and backward completeness are well known notions in abstract interpretation, especially in static program analysis for verifying safety program properties [20,33,29]. The first attempt to weaken the notion of backward-completeness

in abstract interpretation has been defined in [5]. Here the authors introduced the notion of local completeness which corresponds to our definition of local backward-completeness (Definition 3). Partial completeness has been recently introduced as a further weakening of the local completeness property by admitting a limited amount of imprecision measured by a quasi-metric compatible with the underlying abstract domain [11,10].

Besides the partial completeness property, the problem of measuring the imprecision of abstract interpretations is not new. Sotin [42] defines a metric to quantify the result of numerical invariants by calculating the size of the concretization into $\mathbb{R}^n$. This metric can be considered as an instance of pre-metric $\subseteq$-compatible, thus it can be used as distance function for formalizing the partial forward/backward-completeness property of interest.

Crazzolara [25] proposes to substitute partial-orders with quasi-metrics, i.e., the concrete and abstract partially-ordered set turn into quasi-metric spaces. Our approach, instead, preserves the standard abstract interpretation framework and considers the distances as external tools for measuring the incompleteness of abstract operators. A similar idea is proposed by Di Pierro and Wiklicky [41] where partially-ordered domains are replaced by vector spaces lifting abstract interpretation to a probabilistic version where it is possible to apply some well-known distances in linear spaces.

Logozzo et al. [36] adapt the notion of pseudo-metric to be compatible with partially-ordered sets in order to measure the distance between two elements. Their definition of pseudo-metric requires the weak triangle inequality axiom and symmetry, while our definition of pre-metric relaxes those axioms. Moreover, axiom 2 may not be satisfied by pseudo-metrics, therefore their distances may not fit well in our framework.

Casso et al. [13] proposes a list of observations about distance functions when used to measure distances between elements of abstract domains in the context of logic programming. They show that it is possible to induce other distances from one domain to another through the concretization and abstraction functions in a similar way we did in Section 4. However, their notion of distance requires more compatibility with the underlying lattice than our approach as they focus on abstract domains commonly used for analyzing logic programs. For instance, they assume abstract domains to be complete lattices related by a GI with the concrete domains, they require another type of triangle inequality called diamond inequality, and consider distances between comparable elements only. As our notion of pre-metric is weaker than what they require for distances, our framework can be easily instantiated with their distances.

8 Conclusion

We weakened both the local backward-completeness, in presence of a GC, and the local forward-completeness properties in case only a concretization function is available (e.g., the case of convex polyhedra or the domain of formal languages [9]) in order to allow a limited amount of imprecision. This imprecision

is measured according to a distance function formalized as a pre-metric compatible with the underlying pre-order relation. The definition of pre-metrics is general enough to be instantiated by distance functions having different "levels of view". For instance, a distance may be precise enough to satisfies the identity of indiscernibles axiom on the concrete domain, so that it can be used to reason on the local forward-completeness. Different levels of approximation can be obtain by inducing pre-metrics from one domain to another by the use of the concretization or the abstraction maps. Our framework could assist program analysis designers in controlling the propagation of incompleteness, e.g., by choosing the preferred pre-metric according to the imprecision they want to measure and at which level of details, and then by using it for checking how an invariant generated by the analysis grows with respect to the concrete execution or another comparable analysis. This checking process could also be combined with other repairing techniques that aim to enrich the expressiveness of abstract domains [33,6].

Similarly to the other completeness properties [29,4,11,10] both partial forward and backward-completeness properties are undecidable. As a future work we plan to extend the proof system proposed in [11] in order to be able to overestimate, according to δ_D, a bound of incompleteness (either forward or backward) generated by the abstract interpreter without actually executing the program. This, in fact, can be considered as another abstract interpretation analyzing the abstract interpreter [23].

Understanding the propagation of incompleteness through pre-metrics is closely linked to code obfuscation [16], which finds application in software protection [28,30,31] and malware analysis [40,44,26,12]. Being able to quantify the amount of incompleteness induced in the abstract interpretation by a code-obfuscating program transformation could enable us to measure the potency of these transformations. This remains one of the primary open challenges in software protection [43,14,17].

Acknowledgements We wish to thank the anonymous reviewers of SAS 2023 for their detailed comments. This work has been partially supported by the grant PRIN2017 (code: 201784YSZ5) "AnalysiS of PRogram Analyses (ASPRA)".

References

1. https://en.wikipedia.org/wiki/Metric_space#Premetrics
2. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html
3. Arkhangel'Skii, A., Fedorchuk, V.: General topology I: basic concepts and constructions dimension theory, vol. 17. Springer (2012)
4. Bruni, R., Giacobazzi, R., Gori, R., Garcia-Contreras, I., Pavlovic, D.: Abstract extensionality: on the properties of incomplete abstract interpretations. Proc. ACM Program. Lang. **4**(POPL), 28:1–28:28 (2020). https://doi.org/10.1145/3371096

5. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A logic for locally complete abstract interpretations. In: 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021. pp. 1–13. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470608
6. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: Abstract interpretation repair. In: Jhala, R., Dillig, I. (eds.) PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. pp. 426–441. ACM (2022). https://doi.org/10.1145/3519939.3523453
7. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A correctness and incorrectness program logic. J. ACM **70**(2), 15:1–15:45 (2023). https://doi.org/10.1145/3582267
8. Buldygin, V.V., Kozachenko, I.V.: Metric characterization of random variables and random processes, vol. 188. American Mathematical Soc. (2000). https://doi.org/10.1090/mmono/188
9. Campion, M., Dalla Preda, M., Giacobazzi, R.: Abstract interpretation of indexed grammars. In: International Static Analysis Symposium. pp. 121–139. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_7
10. Campion, M., Dalla Preda, M., Giacobazzi, R.: On the properties of partial completeness in abstract interpretation. In: Lago, U.D., Gorla, D. (eds.) Proceedings of the 23rd Italian Conference on Theoretical Computer Science, ICTCS 2022, Rome, Italy, September 7-9, 2022. CEUR Workshop Proceedings, vol. 3284, pp. 79–85. CEUR-WS.org (2022), http://ceur-ws.org/Vol-3284/8665.pdf
11. Campion, M., Dalla Preda, M., Giacobazzi, R.: Partial (in)completeness in abstract interpretation: limiting the imprecision in program analysis. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498721
12. Campion, M., Preda, M.D., Giacobazzi, R.: Learning metamorphic malware signatures from samples. J. Comput. Virol. Hacking Tech. **17**(3), 167–183 (2021). https://doi.org/10.1007/s11416-021-00377-z
13. Casso, I., Morales, J.F., López-García, P., Giacobazzi, R., Hermenegildo, M.V.: Computing abstract distances in logic programs. In: International Symposium on Logic-Based Program Synthesis and Transformation. pp. 57–72. Springer (2019). https://doi.org/10.1007/978-3-030-45260-5_4
14. Ceccato, M., Tonella, P., Basile, C., Falcarin, P., Torchiano, M., Coppens, B., Sutter, B.D.: Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. Empir. Softw. Eng. **24**(1), 240–286 (2019). https://doi.org/10.1007/s10664-018-9625-6
15. Cohen, J., Hickey, T.J.: Two algorithms for determining volumes of convex polyhedra. J. ACM **26**(3), 401–414 (1979). https://doi.org/10.1145/322139.322141
16. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional (2009)
17. Collberg, C.S., Davidson, J.W., Giacobazzi, R., Gu, Y.X., Herzberg, A., Wang, F.: Toward digital asset protection. IEEE Intelligent Systems **26**(6), 8–13 (2011). https://doi.org/10.1109/MIS.2011.106
18. Cousot, P.: Principles of Abstract Interpretation. The MIT Press, Cambridge, Mass. (2021)
19. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the 2nd International Symposium on Programming. pp. 106–130. Dunod, Paris (1976). https://doi.org/10.1145/390019.808314

20. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
21. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Proceedings of the 6th ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979. pp. 269–282. ACM Press (1979). https://doi.org/10.1145/567752.567778
22. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4), 511–547 (1992). https://doi.org/10.1093/logcom/2.4.511
23. Cousot, P., Giacobazzi, R., Ranzato, F.: A^2i: Abstract^2 interpretation. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290355
24. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 84–96. ACM Press (1978). https://doi.org/10.1145/512760.512770
25. Crazzolara, F.: Quasi-metric spaces as domains for abstract interpretation. In: Falaschi, M., Navarro, M., Policriti, A. (eds.) 1997 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'97, Grado, Italy, June 16-19, 1997. pp. 45–56 (1997)
26. Dalla Preda, M., Giacobazzi, R., Debray, S.K.: Unveiling metamorphism by abstract interpretation of code properties. Theor. Comput. Sci. **577**, 74–97 (2015). https://doi.org/10.1016/j.tcs.2015.02.024
27. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at facebook. Commun. ACM **62**(8), 62–70 (2019). https://doi.org/10.1145/3338112
28. Giacobazzi, R.: Hiding information in completeness holes: New perspectives in code obfuscation and watermarking. In: Cerone, A., Gruner, S. (eds.) Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008. pp. 7–18. IEEE Computer Society (2008). https://doi.org/10.1109/SEFM.2008.41
29. Giacobazzi, R., Logozzo, F., Ranzato, F.: Analyzing program analyses. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 261–273. ACM (2015). https://doi.org/10.1145/2676726.2676987
30. Giacobazzi, R., Mastroeni, I.: Making abstract interpretation incomplete: Modeling the potency of obfuscation. In: Miné, A., Schmidt, D. (eds.) Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7460, pp. 129–145. Springer (2012). https://doi.org/10.1007/978-3-642-33125-1_11
31. Giacobazzi, R., Mastroeni, I., Dalla Preda, M.: Maximal incompleteness as obfuscation potency. Formal Aspects Comput. **29**(1), 3–31 (2017). https://doi.org/10.1007/s00165-016-0374-2
32. Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples, and refinements in abstract model-checking. In: Cousot, P. (ed.) Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2126, pp. 356–373. Springer (2001). https://doi.org/10.1007/3-540-47764-0_20

33. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47**(2), 361–416 (2000). https://doi.org/10.1145/333979.333989
34. Helemskii, A.Y.: Lectures and Exercises on Functional Analysis, vol. 233. American Mathematical Soc. (2006). https://doi.org/10.1090/mmono/233
35. Lawrence, J.: Polytope volume computation. Mathematics of Computation **57**(195), 259–271 (Jul 1991). https://doi.org/10.1090/S0025-5718-1991-1079024-2
36. Logozzo, F.: Towards a quantitative estimation of abstract interpretations. In: Workshop on Quantitative Analysis of Software. Microsoft (June 2009), https://www.microsoft.com/en-us/research/publication/towards-a-quantitative-estimation-of-abstract-interpretations/
37. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2053, pp. 155–172. Springer (2001). https://doi.org/10.1007/3-540-44978-7_10
38. Miné, A.: The octagon abstract domain. In: Burd, E., Aiken, P., Koschke, R. (eds.) Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001. p. 310. IEEE Computer Society (2001). https://doi.org/10.1109/WCRE.2001.957836
39. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Programming Languages **4**(3-4), 120–372 (2017). https://doi.org/10.1561/2500000034
40. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: 23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA. pp. 421–430. IEEE Computer Society (2007). https://doi.org/10.1109/ACSAC.2007.21
41. Pierro, A.D., Wiklicky, H.: Measuring the precision of abstract interpretations. In: Lau, K. (ed.) Logic Based Program Synthesis and Transformation, 10th International Workshop, LOPSTR 2000 London, UK, July 24-28, 2000, Selected Papers. Lecture Notes in Computer Science, vol. 2042, pp. 147–164. Springer (2000). https://doi.org/10.1007/3-540-45142-0_9
42. Sotin, P.: Quantifying the precision of numerical abstract domains. Tech. Rep. HAL Id: inria-00457324, INRIA (2010), https://hal.inria.fr/inria-00457324
43. Sutter, B.D., Collberg, C.S., Dalla Preda, M., Wyseur, B.: Software protection decision support and evaluation methodologies (dagstuhl seminar 19331). Dagstuhl Reports **9**(8), 1–25 (2019). https://doi.org/10.4230/DagRep.9.8.1
44. You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: Proceedings of the Fifth International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2010, November 4-6, 2010, Fukuoka Institute of Technology, Fukuoka, Japan (In conjunction with the 3PGCIC-2010 International Conference). pp. 297–300. IEEE Computer Society (2010). https://doi.org/10.1109/BWCCA.2010.85

[u2] Marco Campion, Isabella Mastroeni, Michele Pasqua, **CU**

Abstract Lipschitz Continuity

https://inria.hal.science/hal-04935306
Under Submission

Abstract Lipschitz Continuity

Marco Campion
Inria & ENS Paris | Université PSL, France

Isabella Mastroeni
University of Verona, Italy

Michele Pasqua
University of Verona, Italy

Caterina Urban
Inria & ENS Paris | Université PSL, France

Abstract

We introduce *Abstract Lipschitz Continuity* (ALC), a generalization of standard Lipschitz Continuity, that ensures proportionally bounded differences in the *semantic approximations* of outputs when the *semantic approximations* of inputs differ slightly. ALC distinguishes between two complementary notions of approximation: *quantitative* differences, expressed via pre-metrics, and *qualitative* (or *semantic*) differences, captured through upper closure operators. ALC allows for reasoning about bounded changes in output properties in settings where standard Lipschitz continuity is too restrictive or inapplicable, such as in program analysis and verification, where understanding semantic properties of inputs and outputs is of key importance.

In the specific context of programs, we formally relate ALC to other well-established program properties, including (Partial) Completeness and (Abstract) program Robustness. Notably, we show that ALC is a stronger requirement than Partial Completeness, a consolidated notion modeling precision loss in program analysis.

Finally, we propose a language- and domain-agnostic deductive system, parametric on the quantitative and semantic approximations of interest, for proving the ALC of programs. The goal in designing this deductive system is to track the assumptions required for ALC to ensure a compositional proof.

2012 ACM Subject Classification Theory of computation → Program analysis; Theory of computation → Abstraction; Theory of computation → Program verification

Keywords and phrases Abstract Lipschitz Continuity, Abstract Interpretation, Partial Completeness

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In mathematical analysis, *Lipschitz continuity* is a strong form of uniform continuity for functions computing over metric spaces, which guarantees that changes in the output are bounded proportionally to changes in the input. It finds numerous applications in various areas of mathematics, including analysis, where it ensures uniform continuity and differentiable properties [14], and optimization, where it plays a key role in convergence guarantees for iterative algorithms [32]. In machine learning, it is used to study robustness, stability and convergence of machine learning models, particularly in adversarial settings [15, 23, 24, 39]. Lipschitz continuity is also relevant and interesting for software, notably to reason about robustness of programs that execute on uncertain inputs [8, 9, 10].

The standard definition of Lipschitz continuity requires that both the input and output spaces of a function (e.g., a program) be equipped with metrics, thereby assuming that controlled variation can be meaningfully captured within the structure imposed by these raw spaces. However, this requirement is often too rigid and fails to account for forms of Lipschitz continuity that remain practically relevant in many important applications. In particular, small changes in the raw representation may correspond to negligible or even irrelevant *semantic* differences. Thus, the lack of a controlled function variation with respect to the raw inputs does not preclude the possibility of a meaningful controlled variation: variations may still be well-behaved when viewed through the lens of *semantic properties* of the inputs and outputs (cf. Ex. 11). This is particularly relevant in many applications, including machine learning [1, 19, 25] and program analysis and verification [13, 18, 36, 37], where the focus often lies on the *semantic properties* of program inputs and outputs.

Our Contribution. To address these limitations, we introduce *Abstract Lipschitz Continuity* (ALC), which ensures that small differences in *semantic approximations* of inputs lead to proportionally bounded differences in the *semantic approximations* of outputs.

We formalize semantic (or *qualitative*) approximations as *upper closure operators*, which are also used in the abstract interpretation framework to model domain abstractions [11, 12]. Values (e.g., character strings) are approximated by considering all other values sharing the same semantic property (e.g., length), admitting an error semantically related to the data. In other words, qualitative or semantic approximations add noise in the *meaning* of what is approximated (cf. Sec. 3).

Abstract Lipschitz Continuity combines these semantic approximations with *quantitative* approximations through their distance in general *pre-metric spaces* [7], i.e., not restricted to metric spaces as the standard definition of Lipschitz continuity (cf. Sec. 3).

We relate ALC for programs to other important and well-studied properties such as (Partial) Completeness in abstract interpretation [4, 7, 20] which limits the imprecision of program analysis (cf. Sec. 4), as well as (Abstract) Robustness [19] in machine learning (cf. Sec. 6). Notably, we show that any abstract Lipschitz continuous function is 0-partial complete, meaning that it does not introduce imprecision — relative to the chosen distance function — when computations are performed over semantic approximations of inputs.

Finally, we propose a novel deductive system for verifying ALC for programs, which is parametric with respect to the chosen input and output semantic approximations and distance functions (cf. Sec. 5). As a particular instance of our general deductive system, when no input and output semantic approximation is performed (i.e., the input and output semantic approximation functions are the identify functions), we find the deductive system proposed by Chaudhuri et al. [9, 10] for proving program robustness.

	pre-	quasisemi-	pseudosemi-	semi-	quasi-	pseudo-	metric
(*non-negativity*)	✓	✓	✓	✓	✓	✓	✓
(*if-identity*)	✓	✓	✓	✓	✓	✓	✓
(*iff-identity*)	✗	✓	✗	✓	✓	✗	✓
(*symmetry*)	✗	✗	✓	✓	✗	✓	✓
(*triangle-inequality*)	✗	✗	✗	✗	✓	✓	✓
Example		$\delta_{\subseteq}$	$\delta_{pat}^{\mathsf{Int}}$	δ_{pat}		$\delta_{siz}, \delta_{\Sigma}$	δ_2
Reference		Ex. 20	Ex. 7	Ex. 7		Ex. 3, 11	Ex. 2

Figure 1 Metrics and their weakening.

2 Preliminaries

We review key preliminaries on metrics, Lipschitz continuity, and abstract interpretation.

Distances. Let $\mathbb{R}^\infty$ be the set of real numbers extended with the infinite symbol ∞, such that for all $r \in \mathbb{R}$, $r < \infty$. Let $\mathbb{R}_{\geq n}$ be the restriction of $\mathbb{R}$ to values greater or equal than $n \in \mathbb{N}$. For instance, $\mathbb{R}^\infty_{\geq 0} \stackrel{\text{def}}{=} \{r \in \mathbb{R} \mid r \geq 0\} \cup \{\infty\}$.

▶ **Definition 1** (Metric). *Given a non-empty set L, a* metric *is a binary function $\delta : L \times L \to \mathbb{R}^\infty$ with the following properties $\forall x, y, z \in L$:*

(1) $\delta(x, y) \geq 0$; *(non-negativity)*

(2) $x = y \;\Leftrightarrow\; \delta(x, y) = 0$; *(iff-identity)*

(3) $\delta(x, y) = \delta(y, x)$; *(symmetry)*

(4) $\delta(x, y) \leq \delta(x, z) + \delta(z, y)$. *(triangle-inequality)*

The pair $\langle L, \delta \rangle$ is called a metric space.

▶ **Example 2** (Euclidean Distance). Consider the set of real numbers $\mathbb{R}$. We define the distance δ_2 between two real values $x, y \in \mathbb{R}$ as the absolute value of their difference, i.e., $\delta_2(x, y) \stackrel{\text{def}}{=} |x - y|$. This is the one-dimensional Euclidean distance, well-known to be a metric.

Due to their axioms, metrics are among the strongest types of distances. As we will see in the next sections, depending on what kind of data we want to measure and its abstraction, a distance may not satisfy one or more metric axioms.

A metric that does not satisfy symmetry is called a *quasi-metric*, while a metric that does not satisfy the $\Leftarrow$ implication of (*iff-identity*) is called a *pseudo-metric*. Semi-metrics satisfy all the axioms except for the triangle inequality. The function δ is called a *pre-metric* if it only satisfies (*non-negativity*) and the $\Rightarrow$ implication of the (*iff-identity*), i.e., the (*if-identity*) axiom. All the other metric axioms are not required, making the definition of pre-metric one of the weakest possible distance function. By composing the words pseudo-, quasi- and semi- we obtain different distance flavors by simply keeping the axioms that are satisfied by all the combined words. For instance, a quasisemi-metric is a pre-metric that additionally satisfies the (*iff-identity*), while a pseudosemi-metric only satisfies (*symmetry*) other than (*if-identity*). Fig. 1 summarizes the above distance notions and their properties. The last two rows display the distance symbol and the example in which the distance is defined and used for the first time. We will occasionally use the subscript δ_{L} in cases where the set L may not be immediately clear from the context. The same convention will be adopted to orderings $\preceq$. From this point forward, whenever we say that a function δ is a distance, we assume that it satisfies, at least, the axioms of a pre-metric.

▶ **Example 3** (Size Distance). Consider the powerset $\wp(L)$ of a set L. We write $size(S)$ for the number of elements in the set $S \in \wp(L)$. We define the distance $\delta_{siz} \colon \wp(L) \times \wp(L) \to \mathbb{R}^{\infty}$ between two sets $S_1, S_2 \in \wp(L)$ as the absolute value of the difference in their size, i.e., $\delta_{siz}(S_1, S_2) \stackrel{\text{def}}{=} |size(S_2) - size(S_1)|$. Note that δ_{siz} is a pseudo-metric since it does not satisfy the (*iff-identity*) axiom: two sets may have the same size yet being different.

Lipschitz Continuity. In mathematical analysis, Lipschitz continuity is a strong form of uniform continuity of functions that establishes a quantitative relationship between changes to the inputs of a function and its outputs. Specifically, it imposes that perturbations to the inputs of a function lead to at most proportional changes to its outputs. The standard definition of Lipschitz continuity assumes that both the input and output domains are metric spaces.

▶ **Definition 4** (Lipschitz Continuity). *Let $\langle C, \delta_{\mathsf{C}} \rangle$ and $\langle D, \delta_{\mathsf{D}} \rangle$ be metric spaces. Let $k \in \mathbb{R}_{\geq 0}$. A function $f : C \to D$ satisfies* k-Lipschitz continuity *w.r.t. $\langle \delta_{\mathsf{C}}, \delta_{\mathsf{D}} \rangle$ if and only if:*

$$\forall x, y \in C \colon \delta_{\mathsf{D}}(f(x), f(y)) \;\leq\; k\delta_{\mathsf{C}}(x, y).$$

A function f satisfies Lipschitz continuity *w.r.t. $\langle \delta_{\mathsf{C}}, \delta_{\mathsf{D}} \rangle$ if and only if there exists $k \in \mathbb{R}_{\geq 0}$ such that f satisfies k-Lipschitz continuity w.r.t. $\langle \delta_{\mathsf{C}}, \delta_{\mathsf{D}} \rangle$.*

The Lipschitz constant k provides an upper bound on the rate of change for the output of the function f, i.e., ${}^{\delta_{\mathsf{D}}(f(x),f(y))}/_{\delta_{\mathsf{C}}(x,y)} \leq k$. Note that, k-Lipschitz continuity can be equivalently formulated as follows:

$$\forall x, y \in C \colon \forall \varepsilon' \geq 0 \colon \delta_{\mathsf{C}}(x, y) \leq \varepsilon' \Rightarrow\; \delta_{\mathsf{D}}(f(x), f(y)) \leq k\varepsilon'$$

Abstract Interpretation. Abstract interpretation [11] provides a general framework for approximating functions by interpreting them over an abstract domain A rather than their exact concrete domain C. It is particularly useful in settings where exact computations are infeasible: decidability is obtained in exchange of an unavoidable information loss. We thus say that A is an *abstraction* of C. Abstractions, originally defined using Galois insertions [11], can equivalently be expressed in terms of upper closure operators [12] (ucos or closures, for short), a formulation we adopt in this work.

▶ **Definition 5** (Upper Closure Operator). *An* upper closure operator *(uco) on a partially ordered set (poset, for short) $\langle C, \preceq \rangle$ is a function $\rho \colon C \to C$ with the following properties $\forall c, c' \in C$:*

(i) $c \preceq c' \Rightarrow\; \rho(c) \preceq \rho(c')$; *(monotonicity)*

(ii) $c \preceq \rho(c)$; *(extensivity)*

(iii) $\rho(\rho(c)) = \rho(c)$. *(idempotence)*

A key property of closures is that they are uniquely determined by the set of their fixpoints $\rho(C) = \{c \in C \mid \rho(c) = c\}$. The set of all upper closure operators on C is denoted by $uco(C)$. As an example, the closure $\mathsf{Sign} \in uco(\wp(\mathbb{Z}))$ abstracts a set of integers by discarding all information except the sign of its values, except when the set contains only the value 0. The closure is defined by the set of fixpoints:

$$\mathsf{Sign}(\wp(\mathbb{Z})) \stackrel{\text{def}}{=} \{\varnothing, \{0\}, \{z \in \mathbb{Z} \mid z \leq 0\}, \{z \in \mathbb{Z} \mid z \geq 0\}, \mathbb{Z}\}$$

3 Abstract Lipschitz Continuity

Semantic and Quantitative Approximations. In many domains, approximations are a fundamental tool for simplifying reasoning while preserving essential properties. Broadly, we can distinguish between *qualitative* (or *semantic*) approximations, and *quantitative* approximations.

Qualitative approximations preserve *properties* of the approximated data. For instance, let $\mathsf{Int}\colon \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ be the function that transforms a set of integers $S \in \wp(\mathbb{Z})$ into the smallest interval $[l, u] \stackrel{\text{def}}{=} \{i \in \mathbb{Z} \mid l \leq i \leq u\}$ that contains it, namely such that $S \subseteq [l, u]$, where $l \in \mathbb{Z} \cup \{-\infty\}$, $u \in \mathbb{Z} \cup \{+\infty\}$ and $l \leq u$. So, for instance, the set of integers $\{0, 1, 4\}$ can be semantically approximated by the interval $[0, 4]$ through Int. More formally, qualitative approximations can be modeled using upper closure operators (e.g., $\mathsf{Int} \in uco(\wp(\mathbb{Z}))$). Given a poset $\langle C, \preceq \rangle$ and $\rho \in uco(C)$, an element $x \in C$ is semantically approximated by $\rho(x)$, and the set $\{y \in C \mid \rho(y) = \rho(x)\}$ represents all elements in C sharing the same semantic approximation as x. Continuing the example, the set {{0,4}, {0,1,4}, {0,2,4}, {0,3,4}, {0,1,2,4}, {0,1,3,4}, {0,1,2,3,4}} contains all the sets of integers S such that $\mathsf{Int}(S) = [0, 4]$.

Quantitative approximations preserve *closeness* of the approximated data, typically measured using a distance function in a suitable topological space. More formally, given a pre-metric space $\langle C, \delta \rangle$ and a fixed constant $\varepsilon \in \mathbb{R}^{\infty}_{\geq 0}$, an element $x \in C$ is quantitatively approximated by any element in the set $\{y \in C \mid \delta(x, y) \leq \varepsilon\}$. For instance, using the size distance δ_{siz} (cf. Ex. 3), we may approximate the set $\{0, 1, 4\}$ by any set of integers whose maximum distance from it is at most $\varepsilon = 1$, e.g. by the set $\{0, 1\}$ or $\{5, 6, 8, 10\}$.

By *combining* the two forms of approximation, we obtain a general approximation that incorporates a quantitative error within a qualitative abstraction, while still keeping the two types of approximations distinct. Let $\langle C, \preceq \rangle$ be a poset and $\langle C, \delta \rangle$ be a pre-metric space, and let $\rho \in uco(C)$ be an abstraction. We define $\delta^{\rho}\colon C \times C \to \mathbb{R}^{\infty}_{\geq 0}$ as:

$$\delta^{\rho}(x, y) \stackrel{\text{def}}{=} \delta(\rho(x), \rho(y))$$

that is, δ^{ρ} calculates the distance between the semantic approximations of x and y with ρ. Clearly, when considering the identity function $id \in uco(C)$ as abstraction (i.e., $\forall x \in C.\ id(x) \stackrel{\text{def}}{=} x$), it holds that $\delta^{id}(x, y) = \delta(x, y)$ for any $x, y \in C$. Note that even if the distance δ satisfies the (*iff-identity*) axiom (thus qualifying as a quasisemi-metric), the derived distance δ^{ρ} may no longer satisfy this axiom due to the input approximation introduced by ρ. This observation also highlights why requiring metric spaces in Def. 4 would be overly restrictive when aiming to define a distance that accounts for both forms of approximation. Nevertheless, δ^{ρ} remains a pre-metric.

▶ **Proposition 6.** *Let $\langle C, \preceq \rangle$ be a poset and let $\rho \in uco(C)$. If $\langle C, \delta \rangle$ is a pre-metric space, then $\langle C, \delta^{\rho} \rangle$ is also a pre-metric space.*

▶ **Example 7** (Path-Length Distance)**.** Let us consider the poset $\langle \wp(\mathbb{Z}), \subseteq \rangle$ and the closure $\mathsf{Int} \in uco(\wp(\mathbb{Z}))$. We define the path-length distance $\delta_{pat}\colon \wp(\mathbb{Z}) \times \wp(\mathbb{Z}) \to \mathbb{N}^{\infty}$ as follows: $\delta_{pat}(S_1, S_2) \stackrel{\text{def}}{=} k$ with $k \in \mathbb{N}$ if $S_1 \subseteq S_2 \vee S_2 \subseteq S_1$ and S_2 has k more elements than S_1 or viceversa. For all other cases, the distance is ∞. So, for instance, $\delta_{pat}(\{0, 1, 4\}, \{0, 1, 4, 10\}) = \delta_{pat}(\{0, 1, 4, 10\}, \{0, 1, 4\}) = 1$ because $\{0, 1, 4, 10\}$ has one more integer than $\{0, 1, 4\}$ namely the number 10, while $\delta_{pat}(\{0, 1, 4\}, \{1, 4, 10\}) = \infty$ because both $\{0, 1, 4\} \not\subseteq \{1, 4, 10\}$ and $\{0, 1, 4\} \not\supseteq \{1, 4, 10\}$ hold. Note that δ_{pat} may differ from δ_{siz} even between comparable sets: $\delta_{pat}(\mathbb{Z}_{>0}, \mathbb{Z}_{\geq 0}) = 1 \neq \infty = \delta_{siz}(\mathbb{Z}_{>0}, \mathbb{Z}_{\geq 0})$. In fact, $\langle \wp(\mathbb{Z}), \subseteq \rangle$ can be seen as a weighted graph where each edge has weight 1 and it connects two sets S_1, S_2 such that

(a) Controlled input/output semantic approximations. **(b)** Suppression of input semantic approximation.

Figure 2 Abstract Lipschitz Continuity.

$S_1 \subset S_2 \vee S_2 \subset S_1$ and there is no other set S' such that $S_1 \subset S' \subset S_2 \vee S_2 \subset S' \subset S_1$. Then the distance $\delta_{pat}(S_1, S_2)$ corresponds to the minimum weighted path between S_1 and S_2. The pair $\langle \wp(\mathbb{Z}), \delta_{pat} \rangle$ forms a semi-metric space. It is not a metric space because δ_{pat} does not satisfy the triangle-inequality axiom: $\delta_{pat}(\{0,1,4\},\{1,4,10\}) = \infty \not\leq 2 = 1+1 = \delta_{pat}(\{0,1,4\},\{0,1,4,10\}) + \delta_{pat}(\{0,1,4,10\},\{1,4,10\})$.

By considering the interval abstraction $\mathsf{Int} \in uco(\wp(\mathbb{Z}))$, we can combine the two forms of approximation, namely δ_{pat} and Int, into $\delta^{\mathsf{Int}}_{pat}$: this new distance calculates the number of more elements between two interval abstractions rather than considering the original input sets. Note that $\delta^{\mathsf{Int}}_{pat}$ loses the (*iff-identity*) axiom as one interval might represent more than one set in $\wp(\mathbb{Z})$, thus $\langle \wp(\mathbb{Z}), \delta^{\mathsf{Int}}_{pat} \rangle$ forms a pseudosemi-metric space.

We can now formally define general approximations.

▶ **Definition 8** (General Approximation). *Let $\langle C, \preceq \rangle$ be a poset and $\langle C, \delta \rangle$ be a pre-metric space, and let $\rho \in uco(C)$. An element $x \in C$ is semantically approximated with ρ* and *quantitatively approximated by δ, up to $\varepsilon \in \mathbb{R}^{\infty}_{\geq 0}$, by any element in the set $\{y \in C \mid \delta^{\rho}(x,y) \leq \varepsilon\}$.*

Continuing Ex. 7, the set $\{0,1,4\}$ can be semantically and quantitatively approximated by $\delta^{\mathsf{Int}}_{pat}$ and $\varepsilon = 1$ in any set in

$$\{S \in \wp(\mathbb{Z}) \mid \delta^{\mathsf{Int}}_{pat}(\{0,1,4\}, S) \leq 1\} = \{S \in \wp(\mathbb{Z}) \mid \mathsf{Int}(S) = [-1,4] \vee \mathsf{Int}(S) = [0,5]\}$$

Abstract Lipschitz Continuity. When approximations are introduced to the inputs of a function (e.g., a program), they propagate through its computations, affecting the output. Understanding how approximations evolve during computations provides insight into the behavior of the function (e.g., the program).

Abstract Lipschitz Continuity (ALC) imposes a *controlled* (linear) error propagation from a *general approximation of the inputs* to the *general approximation of the result of a function computation* (cf. Fig. 2a).

▶ **Definition 9** (Abstract Lipschitz Continuity). *Let $\langle C, \preceq_{\mathsf{C}} \rangle$ and $\langle D, \preceq_{\mathsf{D}} \rangle$ be the input and output domains (posets), respectively. Let $\langle C, \delta_{\mathsf{C}} \rangle$ and $\langle D, \delta_{\mathsf{D}} \rangle$ be pre-metric spaces. Let $\eta \in uco(C)$ and $\rho \in uco(D)$ be the abstractions of the input and output domains, respectively, and $k \in \mathbb{R}_{\geq 0}$. A function $f \colon C \to D$ satisfies* Abstract k-Lipschitz Continuity *(k-ALC, for short) w.r.t. $\langle \delta^{\eta}_{\mathsf{C}}, \delta^{\rho}_{\mathsf{D}} \rangle$ when:*

$$\forall x, y \in C.\ \delta^{\rho}_{\mathsf{D}}(f(x), f(y)) \leq k\delta^{\eta}_{\mathsf{C}}(x,y)$$

A function f satisfies Abstract Lipschitz Continuity *(ALC) if and only if there exists $k \in \mathbb{R}_{\geq 0}$ such that f satisfies Abstract k-Lipschitz Continuity.*

When k-ALC holds, the constant k will be called the *abstract Lipschitz constant*.

Note the difference between Def. 4 of Lipschitz Continuity, and Def. 9 of Abstract Lipschitz Continuity. The former states that the quantitative (metric) distance between two function outputs is at most k times the quantitative (metric) distance between the inputs. The latter captures that the quantitative distance between the semantic approximations (i.e., the properties) of two function outputs ($\delta^{\rho}_{\mathsf{D}}(f(x), f(y))$) is at most k times the quantitative distance between the semantic approximations of the inputs ($\delta^{\eta}_{\mathsf{C}}(x, y)$). The two definitions naturally coincide when both $\langle C, \delta_{\mathsf{C}}\rangle$ and $\langle D, \delta_{\mathsf{D}}\rangle$ are metric-spaces, and the input and output domain abstractions introduce no semantic approximation, namely when $\eta = \rho = id$. In this specific scenario, requiring Lipschitz Continuity w.r.t. $\langle \delta_{\mathsf{C}}, \delta_{\mathsf{D}}\rangle$ is equivalent to requiring ALC w.r.t. $\langle \delta^{id}_{\mathsf{C}}, \delta^{id}_{\mathsf{D}}\rangle$. This also explains why Def. 9 is a generalization of Def. 4 when the input and output domains are considered as posets.

Abstract 0-Lipschitz Continuity represents another special case in which the function computation completely suppresses the input property approximation (cf. Fig. 2b).

Similarly to the concrete definition of k-Lipschitz Continuity (cf. Def. 4), k-ALC can be equivalently reformulated as follows:

▶ **Proposition 10.** *Consider the premises of Def. 9. A function $f\colon C \to D$ satisfies k-ALC w.r.t. $\langle \delta^{\eta}_{\mathsf{C}}, \delta^{\rho}_{\mathsf{D}}\rangle$ if and only if: $\forall x, y \in C.\ \forall \varepsilon' \geq 0.\ \delta^{\eta}_{\mathsf{C}}(x, y) \leq \varepsilon' \ \Rightarrow\ \delta^{\rho}_{\mathsf{D}}(f(x), f(y)) \leq k\varepsilon'$.*

▶ **Example 11.** Let Σ be a chosen alphabet (finite set of characters) and let Σ^* be the Kleene closure of Σ, i.e., the set of all strings of finite length over Σ. We write $\mathit{length}(w)$ to denote the length of the string $w \in \Sigma^*$. We consider the poset $\langle \wp(\Sigma^*), \subseteq\rangle$ and the semantic property $\mathsf{Prefix} \in \mathit{uco}(\wp(\Sigma^*))$ which approximates a set $W \in \wp(\Sigma^*)$ of finite strings with the set of all prefixes of at least one string in W: $\mathsf{Prefix}(W) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid \exists w' \in \Sigma^*\colon ww' \in W\}$. We define $\delta_{\Sigma}\colon \wp(\Sigma^*) \times \wp(\Sigma^*) \to \mathbb{N}^{\infty}$ to compute the absolute difference between the number of string lengths in W_1 and W_2, namely:

$$\delta_{\Sigma}(W_1, W_2) \stackrel{\text{def}}{=} \delta_{siz}(\{\mathit{length}(w_1) \mid w_1 \in W_1\}, \{\mathit{length}(w_2) \mid w_2 \in W_2\})$$

where δ_{siz} is the size distance of Ex. 3, forming the pseudo-metric space $\langle \wp(\Sigma^*), \delta_{\Sigma}\rangle$. Given $|W_1| = n_1$ (e.g., $W_1 = \{a\}$, with $n_1 = 1$) and $|W_2| = n_2$ (e.g., $W_2 = \{bb, ccc\}$, with $n_2 = 2$) with, w.l.g., $n_2 \geq n_1$, then in the worst case all strings in both sets have different lengths, therefore in general $\delta_{\Sigma}(W_1, W_2) \leq n_2 - n_1$ (in the example $\delta_{\Sigma}(W_1, W_2) = 2 - 1 = 1$). The function $f\colon \wp(\Sigma^*) \to \wp(\Sigma^*)$ defined as $f(W) \stackrel{\text{def}}{=} \{w_1 w_2 \mid w_1, w_2 \in W\}$ concatenates all pairs of strings in W. In the example, $f(W_1) = \{aa\}$, while $f(W_2) = \{bbbb, bbccc, cccbb, cccccc\}$. We can observe that, in the worst case, in $f(W_i)$ we have $\frac{1}{2}n_i(n_i + 1)$ different lengths (the 2 factor division comes from the fact $|w_1 w_2| = |w_2 w_1|$). In the example, we do have the worst case, having $\frac{1}{2}n_2(n_2 + 1) = 3$ different lengths. Then we can show that $\delta_{\Sigma}(f(W_1), f(W_2)) \leq \frac{1}{2}(n_2 + n_1 + 1)(n_2 - n_1)$, which implies that f cannot satisfy ALC w.r.t. $\langle \delta^{id}_{\Sigma}, \delta^{id}_{\Sigma}\rangle$ since, in the worst case, the distance $\delta_{\Sigma}(f(W_1), f(W_2))$ increases the distance $\delta_{\Sigma}(W_1, W_2)$ by a factor $(\frac{1}{2}(n_2 + n_1 + 1))$ which is not constant, as Lipschitz continuity would require.

Consider now $\delta^{\mathsf{Prefix}}_{\Sigma}$, which adds all strings of smaller lengths up to the maximum length present in the set. Then, if $l_1 = \max\{\mathit{length}(w) \mid w \in W_1\}$ and $l_2 = \max\{\mathit{length}(w) \mid w \in W_2\}$, we have $\delta^{\mathsf{Prefix}}_{\Sigma}(W_1, W_2) \leq l_2 - l_1$ (supposing w.l.g., $l_2 \geq l_1$). By definition, the longest string in $f(W_i)$ has length $2l_i$, therefore, in general, we have

$$\delta^{\mathsf{Prefix}}_{\Sigma}(f(W_1), f(W_2)) \leq 2l_2 - 2l_1 \leq 2\delta^{\mathsf{Prefix}}_{\Sigma}(W_1, W_2)$$

which shows that f satisfies 2-ALC w.r.t. $\langle \delta^{\mathsf{Prefix}}_{\Sigma}, \delta^{\mathsf{Prefix}}_{\Sigma}\rangle$.

$$\mathsf{a} \in \mathsf{AExp},\ x \in \mathbb{X},\ \mathsf{b} \in \mathsf{BExp}$$
$$\mathsf{Stm} \ni \mathsf{c} ::= \mathbf{skip} \mid x := \mathsf{a} \mid \mathsf{b}?$$
$$\mathsf{Prog} \ni \mathsf{P} ::= \mathsf{c} \mid \mathsf{P}\,;\,\mathsf{P} \mid \mathsf{P} \oplus \mathsf{P} \mid \mathsf{P}^*$$

(a) Language syntax.

$$[\![\mathsf{P}_1\,;\,\mathsf{P}_2]\!]c \stackrel{\text{def}}{=} [\![\mathsf{P}_2]\!] \circ [\![\mathsf{P}_1]\!]c$$
$$[\![\mathsf{P}_1 \oplus \mathsf{P}_2]\!]c \stackrel{\text{def}}{=} [\![\mathsf{P}_1]\!]c \vee [\![\mathsf{P}_2]\!]c$$
$$[\![\mathsf{P}^*]\!]c \stackrel{\text{def}}{=} \bigvee\{[\![\mathsf{P}]\!]^n c \mid n \in \mathbb{N}\}$$

(b) Semantics of programs.

Figure 3 Syntax and semantics of Prog.

4 Abstract Lipschitz Continuity for Programs

Up to this point, the ALC notion has been defined for generic functions. In this section, we focus on two specific aspects: (1) the application of ALC to programs, in particular, to functions representing (monotone) semantics of programs; and (2) a comparison between ALC and the notion of (Partial) Completeness in Abstract Interpretation, a well-established property used to characterize precision loss in program analysis.

Programs. In the following, we will consider programs written in the language Prog of regular commands [3, 33], which is general enough to cover deterministic imperative languages [3] as well as other programming paradigms that include, e.g., non-deterministic and probabilistic computations. The syntax of the language is given in Fig. 3a, where $\oplus$ denotes non-deterministic choice and * is the Kleene closure. We completed the grammar in [3] with an explicit grammar for basic commands in Stm (**skip**, variable assignments, Boolean tests), and we assume a standard grammar for arithmetic expressions in AExp and Boolean expressions in BExp. Variables x range from a denumerable set $\mathbb{X}$ while values v range from a denumerable set $\mathbb{V}$ (e.g., integer or natural numbers).

We assume to have a concrete monotone semantics $[\![\mathsf{c}]\!]\colon C \to C$ for basic commands $\mathsf{c} \in \mathsf{Stm}$ on the complete lattice $\langle C, \preceq, \vee, \wedge, \top, \bot\rangle$, where $\preceq$ is the partial order, $\vee$ is the least upper bound (lub), $\wedge$ the greatest lower bound (glb), $\top$ is the supremum of C and $\bot$ is the infimum of C. Then, the *concrete semantics* $[\![\cdot]\!]\colon \mathsf{Prog} \to C \to C$ for programs is inductively defined on program syntax as in Fig 3b. It is easy to note that, for any program $\mathsf{P} \in \mathsf{Prog}$, $[\![\mathsf{P}]\!]$ is monotone by construction. Given a program $\mathsf{P} \in \mathsf{Prog}$, the semantics $[\![\mathsf{P}]\!]$ is said to be additive when it preserves arbitrary joins, i.e., $\forall S \subseteq C.\ \bigvee\{[\![\mathsf{P}]\!]c \mid c \in S\} = [\![\mathsf{P}]\!](\bigvee S)$.

▶ **Example 12** (Collecting semantics)**.** As an example, consider the complete lattice $\langle \wp(\mathbb{M}), \subseteq, \cup, \cap, \mathbb{M}, \varnothing\rangle$ of program memories, where a program memory $\mathtt{m} \in \mathbb{M}$ is a function mapping variables to values, namely $\mathtt{m}\colon \mathbb{X} \to \mathbb{V}$. We can define a collecting big-step semantics $[\![\mathsf{P}]\!]\colon \wp(\mathbb{M}) \to \wp(\mathbb{M})$ for a program $\mathsf{P} \in \mathsf{Prog}$ as the standard predicate transformer semantics on sets of program memories $\wp(\mathbb{M})$. Assume a big-step evaluation semantics $\Downarrow_{\mathsf{a}}$ for arithmetic expressions and $\Downarrow_{\mathsf{b}}$ for Boolean expressions. Given a set of program memories $S \in \wp(\mathbb{M})$, the semantics of basic commands is defined as:

$$[\![\mathbf{skip}]\!]S \stackrel{\text{def}}{=} S$$
$$[\![x := \mathsf{a}]\!]S \stackrel{\text{def}}{=} \{\mathtt{m}[x \leftarrow\!\shortmid v] \mid \mathtt{m} \in S \wedge \mathtt{m} \Downarrow_{\mathsf{a}} v\}$$
$$[\![\mathsf{b}?]\!]S \stackrel{\text{def}}{=} \{\mathtt{m} \in S \mid \mathtt{m} \Downarrow_{\mathsf{b}} \mathtt{tt}\}$$

The collecting semantics for basic commands is monotone by construction on the powerset lattice of memories, and so $[\![\mathsf{P}]\!]$ is also monotone for any program $\mathsf{P} \in \mathsf{Prog}$. Moreover, it is easy to note that $[\![\mathsf{P}]\!]\colon \wp(\mathbb{M}) \to \wp(\mathbb{M})$ satisfies additivity as well.

We are now ready to instantiate Def. 9, originally stated for generic functions, to the specific case of monotone semantic functions by employing the program semantics of interest together with a chosen distance. Let $\langle C, \preceq \rangle$ be a poset and $\langle C, \delta \rangle$ a pre-metric space over the same domain. Let $\eta, \rho \in uco(C)$ be the input and output abstractions, respectively, and $k \in \mathbb{R}_{\geq 0}$. Consider a monotone program semantics $[\![\cdot]\!] \colon \mathsf{Prog} \to C \to C$. We say that *the semantics* $[\![\mathsf{P}]\!]$ *of a program* $\mathsf{P} \in \mathsf{Prog}$ satisfies Abstract k-Lipschitz Continuity w.r.t. $\langle \delta^\eta, \delta^\rho \rangle$:

$$\forall c_1, c_2 \in C.\ \delta^\rho([\![\mathsf{P}]\!]c_1, [\![\mathsf{P}]\!]c_2) \leq k\delta^\eta(c_1, c_2)$$

(Partial) Completeness. Given a monotone function $f \colon C \to D$ over posets $\langle C, \preceq_{\mathtt{C}} \rangle$ and $\langle D, \preceq_{\mathtt{D}} \rangle$ (such as the collecting big-step semantics $[\![\mathsf{P}]\!] \colon \wp(\mathbb{M}) \to \wp(\mathbb{M})$ defined above over $\langle \wp(\mathbb{M}), \subseteq, \cup, \cap, \mathbb{M}, \varnothing \rangle$), the abstractions $\eta \in uco(C)$ and $\rho \in uco(D)$ can be used to approximate computations, thus defining an abstract version $f^\sharp \colon \eta(C) \to \rho(D)$ of f. An abstract function $f^\sharp \colon \eta(C) \to \rho(D)$ is *sound* when $\rho \circ f \preceq_{\mathtt{D}} f^\sharp \circ \eta$ [11]. A sound by construction approximation is $\bar{f} \stackrel{\text{def}}{=} \rho \circ f \circ \eta$, called the *best correct approximation* (bca) [12] of f. Any $f^\sharp$ soundly approximating f is, in fact, equal or less precise than the bca, formally: $\rho \circ f \preceq_{\mathtt{D}} \bar{f} \preceq_{\mathtt{D}} f^\sharp \circ \eta$ [11]. In the following, we will often shorten the composition of functions such as $\rho \circ f \circ \eta$, by $\rho f \eta$.

A sound abstract computation $f^\sharp \colon \eta(C) \to \rho(D)$ performs a precise approximation of a (concrete) monotone function $f \colon C \to D$ whenever $\rho f = f^\sharp \eta$. It has been proved that for a precise abstract approximation to exist, the bca $\bar{f} \stackrel{\text{def}}{=} \rho \circ f \circ \eta$ must also be precise [12, 20]. In particular, if $f^\sharp$ is a precise abstract approximation of f then $f^\sharp = \bar{f}$. *Completeness* [12, 20] in abstract interpretation is a desirable property that ensures the existence of a precise abstract approximation of a (concrete) monotone function f. Proving the completeness of f means proving the bca $\bar{f}$ is precise. Formally,

▶ **Definition 13** (Completeness [12, 20]). *Let $\langle C, \preceq_{\mathtt{C}} \rangle$ and $\langle D, \preceq_{\mathtt{D}} \rangle$ be posets, and let $\eta \in uco(C)$ and $\rho \in uco(D)$ be the input and output abstractions, respectively. A monotone function $f \colon C \to D$ satisfies* Completeness *w.r.t. $\langle \eta, \rho \rangle$ if and only if $\forall x \in C \colon \rho f(x) = \rho f \eta(x)$.*

In practice, Completeness is rarely satisfied. For this reason, Campion et al. [4, 6, 7] introduced a weaker notion of completeness, called *Partial Completeness*, by the use of pre-metrics compatible with the ordering of the underlying poset.

▶ **Definition 14** (Order-Compatible Distance [7]). *Let $\langle L, \preceq \rangle$ be a poset. A distance $\delta : L \times L \to \mathbb{R}^\infty$ is said to be* compatible with the ordering $\preceq$ *or, in short,* $\preceq$-compatible*, if and only if it also satisfies the following property $\forall x, y, z \in L$:*

$$x \preceq y \preceq z \ \Rightarrow\ \delta(x, y) \leq \delta(x, z) \ \wedge\ \delta(y, z) \leq \delta(x, z). \qquad \textit{(chains-order)}$$

A poset $\langle L, \preceq \rangle$ equipped with a $\preceq$-compatible distance δ is called a distance compatible space *and is denoted by the triple $\langle L, \preceq, \delta \rangle$.*

The purpose of the (*chains-order*) axiom is to give a meaning to distances between comparable elements. Notably, let $f_1^\sharp$ and $f_2^\sharp$ be sound abstract approximations of a concrete monotone function $f \colon C \to D$, i.e., $\rho \circ f \preceq_{\mathtt{D}} f_1^\sharp \circ \eta$ and $\rho \circ f \preceq_{\mathtt{D}} f_2^\sharp \circ \eta$. If $f_1^\sharp$ is more precise than $f_2^\sharp$, i.e., $f_1^\sharp \preceq_{\mathtt{D}} f_2^\sharp$, we expect a decrease in the imprecision (distance) with respect to the concrete computation when using $f_1^\sharp$ rather than $f_2^\sharp$, i.e., $\forall x \in D \colon \delta(\rho f(x), f_1^\sharp \eta(x)) \leq \delta(\rho f(x), f_2^\sharp \eta(x))$.

▶ **Example 15.** The poset $\langle \mathbb{R}, \leq \rangle$ equipped with the Euclidean distance δ_2 from Ex. 2 is a metric compatible space. The poset $\langle \wp(L), \subseteq \rangle$ and the size distance δ_{siz} from Ex. 3 form a pseudometric compatible space. In Ex. 7, $\langle \wp(\mathbb{Z}), \subseteq, \delta_{pat}^{\mathsf{Int}} \rangle$ is a pseudosemi-metric compatible space.

Def. 14 is general enough to be instantiated with other definitions of distances used in the literature of abstract interpretation (see, e.g., [4, 26, 27, 35, 38]).

We can now recall the definition of ε-Partial Completeness.

▶ **Definition 16** (ε-Partial Completeness [4, 7]). *Let $\langle C, \preceq_{\mathtt{C}}\rangle$ be a poset and $\langle D, \preceq_{\mathtt{D}}, \delta_{\mathtt{D}}\rangle$ be a pre-metric compatible space, let $\eta \in uco(C)$ and $\rho \in uco(D)$ be the input and output abstractions, respectively. Let $\varepsilon \in \mathbb{R}^{\infty}_{\geq 0}$. A monotone function $f : C \to D$ satisfies* ε-Partial Completeness *w.r.t. $\langle \eta, \delta^{\rho}_{\mathtt{D}}\rangle$ if and only if $\forall x \in C\colon \delta^{\rho}_{\mathtt{D}}(f(x), f\eta(x)) \leq \varepsilon$.*

The equality requirement of Def. 13 is relaxed by admitting a bounded imprecision, i.e. a bounded distance, between $\rho f(x)$ and the bca $\rho f\eta(x)$ for all $x \in C$, which must not exceed ε. The imprecision to be measured and bounded is encoded in the pre-metric $\preceq_{\mathtt{D}}$-compatible $\delta_{\mathtt{D}}$ defined over the output domain D.

▶ **Example 17.** Let $\langle \wp(\mathbb{Z}), \subseteq, \delta_{siz}\rangle$ be an instance of the pseudo-metric compatible space from Ex. 3. Consider the program $\mathsf{M}\colon\ x := x \bmod 2$ and its collecting semantics $[\![\mathsf{M}]\!] : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$. Let $\rho = \eta = \mathsf{Int}$ where $\mathsf{Int} \in uco(\wp(\mathbb{Z}))$ is the interval abstraction defined in Sec. 3. Then $[\![\mathsf{M}]\!]$ does not satisfy Completeness w.r.t. $\langle \mathsf{Int}, \mathsf{Int}\rangle$ because for the input $\{2, 4\}$ we get:

$$\mathsf{Int}([\![\mathsf{M}]\!]\{2,4\}) = [0,0] \subset [0,1] = \mathsf{Int}([\![\mathsf{M}]\!]\{2,3,4\}) = \mathsf{Int}([\![\mathsf{M}]\!]\mathsf{Int}(\{2,4\}))$$

However, if we allow an imprecision quantified by $\varepsilon = 1$, we get:

$$\delta^{\mathsf{Int}}_{siz}([\![\mathsf{M}]\!]\{2,4\}, [\![\mathsf{M}]\!](\mathsf{Int}(\{2,4\}))) = \delta_{siz}([0,0],[0,1]) \leq 1$$

In particular, it is easy to note that $\delta^{\mathsf{Int}}_{siz}([\![\mathsf{M}]\!]S, [\![\mathsf{M}]\!](\mathsf{Int}(S))) \leq 1$, for all sets $S \in \wp(\mathbb{Z})$, which implies that $[\![\mathsf{M}]\!]$ is 1-Partial Complete with respect to $\langle \mathsf{Int}, \delta^{\mathsf{Int}}_{siz}\rangle$.

It is worth noting that, if a function f is proved to satisfy Completeness for abstractions $\langle \eta, \rho\rangle$, then f is also 0-Partial Complete for $\langle \eta, \delta^{\rho}\rangle$ with respect to any pre-metric order-compatible δ (thanks to the (*if-identity*) axiom). However, the converse does not hold if the (*iff-identity*) axiom is not satisfied by δ, e.g., when δ is a pseudo-metric.

Abstract Lipschitz Continuity and Partial Completeness. It turns out that ALC (cf. Def 9) is a much stronger requirement than Partial Completeness (cf. Def. 16) for a program (semantics, or a monotone function). Indeed, satisfying ALC is sufficient to also satisfy 0-Partial Completeness:

▶ **Theorem 18.** *Let $\langle C, \preceq_{\mathtt{C}}, \delta_{\mathtt{C}}\rangle$ and $\langle D, \preceq_{\mathtt{D}}, \delta_{\mathtt{D}}\rangle$ be pre-metric compatible spaces, let $\eta \in uco(C)$, $\rho \in uco(D)$ be abstractions, and let $k \in \mathbb{R}_{\geq 0}$. Consider a monotone function $f : C \to D$. Then, if f satisfies k-ALC w.r.t. $\langle \delta^{\eta}_{\mathtt{C}}, \delta^{\rho}_{\mathtt{D}}\rangle$, it also satisfies 0-Partial Completeness w.r.t. $\langle \eta, \delta^{\rho}_{\mathtt{D}}\rangle$, namely:*

$$[\forall x, y \in C.\ \delta^{\rho}_{\mathtt{D}}(f(x), f(y)) \leq\ k\delta^{\eta}_{\mathtt{C}}(x, y)] \ \Rightarrow\ [\forall x \in C.\ \delta^{\rho}_{\mathtt{D}}(f(x), f\eta(x)) \leq 0]$$

Proofs of the above result, as well as of the corollary below, are provided in Appendix A. Knowing that a monotone function f is k-ALC for $\langle \delta^{\eta}_{\mathtt{C}}, \delta^{\rho}_{\mathtt{D}}\rangle$ leads to conclude that the bca $\rho f\eta$ is 0-partial complete for the same abstractions. Specifically, $\rho f\eta$ will produce *no imprecision* according to $\delta_{\mathtt{D}}$, when used to approximate f.

When $\delta_{\mathtt{D}}$ is a quasisemi-metric, then the above result implies that $\rho f\eta$ is a complete approximation of f thanks to the (*iff-identity*) axiom.

▶ **Corollary 19.** *If $\langle D, \preceq_{\mathtt{D}}, \delta_{\mathtt{D}}\rangle$ is a quasisemi-metric compatible space then k-ALC w.r.t. $\langle \delta^{\eta}_{\mathtt{C}}, \delta^{\rho}_{\mathtt{D}}\rangle$ implies Completeness w.r.t. $\langle \eta, \rho\rangle$.*

▶ **Example 20.** Let R be the following program:

$$(x > 0?\,;\, x := x - 1) \oplus (x \leq 0?\,;\, x := x + 1)$$

which increments all non-negative values by 1 and decrements all non-positive values by 1. Let us consider the program R^*, which is the Kleene closure of R, and its collecting semantics $[\![\mathsf{R}^*]\!] : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$. Let $\langle \wp(\mathbb{Z}), \subseteq, \delta_{\subseteq} \rangle$ be a quasisemi-metric compatible space where, for any two sets $S_1, S_2 \in \wp(\mathbb{Z})$, $\delta_{\subseteq}(S_1, S_2) \stackrel{\text{def}}{=} \delta_{siz}(S_1, S_2)$ (cf. Ex. 3) if $S_1 \subseteq S_2$, ∞ otherwise. Compared to δ_{siz}, the distance $\delta_{\subseteq}$ looses the (*symmetry*) and the (*triangle-inequality*) axioms but gains the (*iff-identity*) axiom. Let us also consider again the interval closure $\mathsf{Int} \in uco(\wp(\mathbb{Z}))$. The collecting semantics $[\![\mathsf{R}^*]\!]$ satisfies 1-ALC w.r.t. $\langle \delta^{\mathsf{Int}}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq} \rangle$. Indeed, $[\![\mathsf{R}^*]\!]$ is monotone by definition, thus preserving the inclusion relation, and either reduces the distance of input intervals or leaves them unchanged. For instance:

$$\delta^{\mathsf{Int}}_{\subseteq}([\![\mathsf{R}^*]\!]([2,6]), [\![\mathsf{R}^*]\!]([0,7])) = \delta^{\mathsf{Int}}_{\subseteq}([0,6],[0,7]) = 1 \leq 3 = \delta^{\mathsf{Int}}_{\subseteq}([2,6],[0,7])$$
$$\delta^{\mathsf{Int}}_{\subseteq}([\![\mathsf{R}^*]\!]([-5,-2]), [\![\mathsf{R}^*]\!]([-7,0])) = \delta^{\mathsf{Int}}_{\subseteq}([-5,1],[-7,1]) = 2 \leq 5 = \delta^{\mathsf{Int}}_{\subseteq}([-5,-2],[-7,0])$$
$$\delta^{\mathsf{Int}}_{\subseteq}([\![\mathsf{R}^*]\!]([-2,3]), [\![\mathsf{R}^*]\!]([-5,3])) = \delta^{\mathsf{Int}}_{\subseteq}([-2,3],[-5,3]) = 3 \leq 3 = \delta^{\mathsf{Int}}_{\subseteq}([-2,3],[-5,3])$$

By Thm. 18, the semantics $[\![\mathsf{R}^*]\!]$ also satisfies 0-Partial Completeness w.r.t. $\langle \mathsf{Int}, \delta^{\mathsf{Int}}_{\subseteq} \rangle$, i.e., $\delta^{\mathsf{Int}}_{\subseteq}([\![\mathsf{R}^*]\!]S, [\![\mathsf{R}^*]\!]\mathsf{Int}(S)) \leq 0$, for any $S \in \wp(\mathbb{Z})$. Moreover, since $\delta_{\subseteq}$ is a quasisemi-metric we can also conclude that $[\![\mathsf{R}^*]\!]$ is complete w.r.t. $\langle \mathsf{Int}, \mathsf{Int} \rangle$, namely, its bca $\mathsf{Int} \circ [\![\mathsf{R}^*]\!] \circ \mathsf{Int}$ does not add any imprecision when approximating $[\![\mathsf{R}^*]\!]$. It is easy to note that 1-ALC w.r.t. $\langle \delta^{\mathsf{Int}}_{\subseteq}, \delta^{\mathsf{Int}}_{pat} \rangle$ also holds for $[\![\mathsf{R}^*]\!]$.

Another way to interpret Cor. 19 (and analogously Thm. 18) is as follows: if a monotone function f does not admit a precise bca $\overline{f}$ over $\langle \eta, \rho \rangle$, then f cannot be ALC for $\langle \delta^{\eta}_{\mathsf{C}}, \delta^{\rho}_{\mathsf{D}} \rangle$, where δ_{C} and δ_{D} are any quasisemi-metric order-compatible distances. This is because Partial Completeness only compares the output results (of ρf and $\rho f \eta$) on the same chain of the poset $\langle D, \preceq_{\mathsf{D}} \rangle$, a consequence of the soundness condition $\rho f \preceq_{\mathsf{D}} \rho f \eta$.

▶ **Example 21.** Consider the pseudo-metric order-compatible space $\langle \wp(\mathbb{Z}), \subseteq, \delta_{siz} \rangle$ and the interval closure $\mathsf{Int} \in uco(\wp(\mathbb{Z}))$. The semantics $[\![\mathsf{R}]\!]$ from Ex. 20 does not satisfy 0-Partial Completeness w.r.t. $\langle \mathsf{Int}, \delta^{\mathsf{Int}}_{siz} \rangle$: given $X = \{-1, 1\}$, we have $\delta^{\mathsf{Int}}_{siz}([\![\mathsf{R}]\!]X, [\![R]\!]\mathsf{Int}(X)) = \delta^{\mathsf{Int}}_{siz}([0,0],[0,1]) = 1 \neq 0$. Thus, $[\![\mathsf{R}]\!]$ cannot satisfy ALC for $\langle \delta^{\mathsf{Int}}_{siz}, \delta^{\mathsf{Int}}_{siz} \rangle$. In fact, it is easy to note that $[\![\mathsf{R}]\!]$ satisfies 1-Partial Completeness for all inputs.

In Section 6 we also relate ALC to other important program properties in the literature.

5 Proving Abstract Lipschitz Continuity for Programs

Deductive systems for the verification of Completeness [16], Partial Completeness [4] and (concrete) Lipschitz continuity [9, 10] properties of programs have already been formalized in the literature. In this section we introduce a novel deductive system, inductively defined on the program's syntax, that is able to soundly prove the new ALC notion of an additive program semantics w.r.t. the input and output abstractions $\langle \eta, \rho \rangle$ and a given pre-metric δ. Our objective in designing this deductive system has been to track the assumptions of ALC needed for having a compositional proof. Soundness here means that when the semantics of a program P is typed as k-ALC w.r.t. $\langle \delta^{\eta}, \delta^{\rho} \rangle$ by the deductive system, then $[\![\mathsf{P}]\!] : C \to C$ is certainly k-ALC for $\langle \delta^{\eta}, \delta^{\rho} \rangle$. Conversely, the deductive system is not complete, namely, not all abstract Lipschitz continuous program semantics proofs can be derived through the

$$\frac{[\![\mathsf{c}]\!] \in k\text{-}Lip\langle\delta^\eta, \delta^\rho\rangle}{k \vdash [\delta^\eta]\,\mathsf{c}\,(\delta^\rho)}\ (\textbf{base})$$

$$\frac{k' \vdash [\delta^{\eta'}]\,\mathsf{P}\,(\delta^{\rho'}) \quad k' \leq k \quad \eta' \in t\text{-}Lip\langle\delta^\eta, \delta^{\eta'}\rangle \quad \rho \in s\text{-}Lip\langle\delta^{\rho'}, \delta^\rho\rangle}{stk \vdash [\delta^\eta]\,\mathsf{P}\,(\delta^\rho)}\ (\textbf{weaken})$$

$$\frac{k_1 \vdash [\delta^\eta]\,\mathsf{P}_1\,(\delta^\rho) \quad k_2 \vdash [\delta^\eta]\,\mathsf{P}_2\,(\delta^\rho) \quad \eta \in t\text{-}Lip\langle\delta^\rho, \delta^\eta\rangle}{k_1 t k_2 \vdash [\delta^\eta]\,\mathsf{P}_1; \mathsf{P}_2\,(\delta^\rho)}\ (\textbf{seq})$$

$$\frac{k_1 \vdash [\delta^\eta]\,\mathsf{P}_1\,(\delta^\rho) \quad k_2 \vdash [\delta^\eta]\,\mathsf{P}_2\,(\delta^\rho) \quad \rho \in t\text{-}Lip\langle\delta^{id}, \delta^\rho\rangle \quad \oplus\text{-Bound}(\langle\delta^\eta, \delta^\rho\rangle, \mathfrak{b})}{t\mathfrak{b}(k_1, k_2) \vdash [\delta^\eta]\,\mathsf{P}_1 \oplus \mathsf{P}_2\,(\delta^\rho)}\ (\textbf{join})$$

$$\frac{k \vdash [\delta^\eta]\,\mathsf{P}\,(\delta^\rho) \quad \eta \in t\text{-}Lip\langle\delta^\rho, \delta^\eta\rangle \quad *\text{-Bound}(\mathsf{P}^*, m)}{(tk)^m \vdash [\delta^\eta]\,\mathsf{P}^*\,(\delta^\rho)}\ (\textbf{star})$$

Figure 4 A deductive system for proving ALC for Prog.

deductive system. This means that we are performing an under-approximation of the set of all abstract Lipschitz continuous program semantics.

We first introduce the following set

$$k\text{-}Lip\langle\delta^\eta, \delta^\rho\rangle \stackrel{\text{def}}{=} \{f \in C \to C \mid f \text{ is Abstract } k\text{-Lipschitz Continuous w.r.t. } \langle\delta^\eta, \delta^\rho\rangle\}$$

of all abstract k-Lipschitz continuous functions on the complete lattice $\langle C, \preceq\rangle$ for $\langle\delta^\eta, \delta^\rho\rangle$. The following lemma outlines some basic properties of this class.

▶ **Lemma 22.** *The following hold for all functions $f \in C \to C$, closures $\eta, \rho \in uco(C)$, pre-metric δ and $k \in \mathbb{R}_{\geq 0}$:*

(i) $k \geq 1 \Rightarrow \rho \in k\text{-}Lip\langle\delta^\rho, \delta^\rho\rangle$

(ii) f *is* k*-Lipschitz continuous w.r.t.* $\langle\delta, \delta\rangle \Leftrightarrow f \in k\text{-}Lip\langle\delta^{id}, \delta^{id}\rangle \wedge \delta$ *metric*

(iii) $\rho \in k\text{-}Lip\langle\delta^{id}, \delta^{id}\rangle \Leftrightarrow \rho \in k\text{-}Lip\langle\delta^{id}, \delta^\rho\rangle$

(i) states that, when considering the same input-output abstractions (i.e. $\eta = \rho$), then the abstraction function is k-ALC for any $k \geq 1$. Moreover, for the statement (ii), when both input-output abstractions are the identity function id and the distance δ is a metric, then the class $k\text{-}Lip\langle\delta^{id}, \delta^{id}\rangle$ corresponds precisely to the set of all (concrete) k-Lipschitz continuous functions (cf. Def. 4). Finally, (iii) shows that, when a closure ρ satisfies ALC w.r.t. $\langle\delta^{id}, \delta^{id}\rangle$, then ρ also satisfies k-ALC for $\langle\delta^{id}, \delta^\rho\rangle$. This is due to the idempotence property of closure operators.

From now on, we fix an additive program semantics of interest $[\![\cdot]\!]\colon \mathsf{Prog} \to C \to C$ as well as the complete lattice $\langle C, \preceq, \vee, \wedge, \top, \bot\rangle$, and we will also use the statement "P is k-ALC w.r.t. $\langle\delta^\eta, \delta^\rho\rangle$" to indicate that the semantics $[\![\mathsf{P}]\!]$ is abstract k-Lipschitz continuous w.r.t. $\langle\delta^\eta, \delta^\rho\rangle$, i.e. $[\![\mathsf{P}]\!] \in k\text{-}Lip\langle\delta^\eta, \delta^\rho\rangle$.

The deductive rules are provided in Fig. 4. The judgments take the form:

$$k \vdash [\delta^\eta]\,\mathsf{P}\,(\delta^\rho)$$

We will later show that deriving a judgment $k \vdash [\delta^\eta]\,\mathsf{P}\,(\delta^\rho)$ through the deductive rules in Fig. 4, implies that $[\![\mathsf{P}]\!] \in k\text{-}Lip\langle\delta^\eta, \delta^\rho\rangle$. Let us examine each rule and provide an intuitive, informal explanation for better understanding.

The rule (**base**) allows to derive the triple $k \vdash [\delta^\eta]\,\mathsf{c}\,(\delta^\rho)$ for all basic transfer functions $\mathsf{c} \in \mathsf{Stm}$ (i.e., for **skip**, assignments and Boolean guards) by assuming that we have a proof of k-ALC of them, encoded by the predicate $[\![\mathsf{c}]\!] \in k\text{-}Lip\langle\delta^\eta, \delta^\rho\rangle$.

The rule (**weaken**) allows to weaken both the abstract Lipschitz constant and the abstractions considered. In particular, when we are able to derive the k'-ALC for program P w.r.t. $\langle\delta^{\eta'}, \delta^{\rho'}\rangle$, then we can always deduce a higher abstract Lipschitz constant $k \geq k'$ without changing the validity of the triple. For the input abstraction η', we can consider a new input abstraction η whenever η' is proved to be t-ALC w.r.t. $\langle\delta^\eta, \delta^{\eta'}\rangle$ with η as input abstraction. This weakening comes at the cost of multiplying the already deduced constant k' with the new constant t. This could happen, for instance, when η is in fact widening the distance $\delta^{\eta'}(c_1, c_2)$ between any two elements $c_1, c_2 \in C$, by a constant factor of t, namely by $t\delta^\eta(c_1, c_2)$. Conversely, we can weaken the output abstraction ρ' by a new abstraction ρ whenever ρ is proved to be ALC for $\langle\delta^{\rho'}, \delta^\rho\rangle$ namely with ρ' as input abstraction. Here ρ could represent a narrow output abstraction in terms of distance δ between elements in C with respect to ρ', namely having distance $\delta^\rho(c_1, c_2) \leq s\delta^{\rho'}(c_1, c_2)$ and thus introducing a new abstract Lipschitz constant s. Note that the rule (**weaken**) allows also for selecting which weakening we want to apply. For instance, if we want to weaken the abstract Lipschitz constant k' only, then we can set $\eta' \in 1\text{-}Lip\langle\delta^{\eta'}, \delta^{\eta'}\rangle$ and $\rho' \in 1\text{-}Lip\langle\delta^{\rho'}, \delta^{\rho'}\rangle$ in the premises as they always hold (cf. statement (i) of Lem. 22) without modifying any abstraction.

Composition of programs is treated by the rule (**seq**). Although it is well known that composing two (concrete) Lipschitz continuous functions f_1 and f_2 with Lipschitz constants k_1 and k_2, respectively, gives as result a new k_1k_2-Lipschitz continuous function $f_2 \circ f_1$, this in general does not always hold for ALC as abstractions come into play. However, when we have a derivation for P_1 and P_2 with abstract Lipschitz constants k_1 and k_2, respectively, and we are able to prove that the input abstraction η is t-ALC w.r.t. $\langle\delta^\rho, \delta^\eta\rangle$, then this is a sufficient condition for deriving the k_2tk_1-ALC of the composition $\mathsf{P}_1;\mathsf{P}_2$. Requiring $\eta \in t\text{-}Lip\langle\delta^\rho, \delta^\eta\rangle$ corresponds to require $\delta^\eta(c_1, c_2) \leq t\delta^\rho(c_1, c_2)$, namely that we have a linear relation between their distances: when $t \geq 1$ then ρ is widening the distance, while when $0 < t < 1$ then ρ is narrowing their distances, both cases with a constant factor of t. Note that, when the input and output abstractions coincide, i.e. $\eta = \rho$, then $\rho \in 1\text{-}Lip\langle\delta^\rho, \delta^\rho\rangle$ holds trivially (cf. statement (i) of Lem. 22). As a consequence, the ALC property is closed under composition, analogously to the standard Lipschitz continuity property.

The rule (**join**) involves the join operator. Similarly for the composition, the join of two ALC functions is not necessarily ALC. The problem here stems in the fact that the resulting abstract Lipschitz constant bound could not be determined by knowing only the abstract Lipschitz constants of both P_1 and P_2. This is because the distance between the execution of $\mathsf{P}_1 \oplus \mathsf{P}_2$ and the join of the two post-conditions, relies on the underlying structure of the input and output abstractions considered. Our solution, inspired by [4, 8], consists of introducing a new predicate $\oplus$-Bound$(\langle\delta^\eta, \delta^\rho\rangle, \mathfrak{b})$ parameterized by a binary function $\mathfrak{b} : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ producing a new abstract Lipschitz constant.

▶ **Definition 23 ($\oplus$-Bound).** *Consider a binary function $\mathfrak{b} : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. The predicate $\oplus$-Bound$(\langle\delta^\eta, \delta^\rho\rangle, \mathfrak{b})$ holds when the function $\mathfrak{b}$ satisfies the following condition for any $\mathsf{P}_1, \mathsf{P}_2 \in \mathsf{Prog}$:*

$$[\![\mathsf{P}_1]\!] \in k_1\text{-}Lip\langle\delta^\eta, \delta^\rho\rangle \text{ and } [\![\mathsf{P}_2]\!] \in k_2\text{-}Lip\langle\delta^\eta, \delta^\rho\rangle \;\Rightarrow\; \rho[\![\mathsf{P}_1]\!] \oplus \rho[\![\mathsf{P}_2]\!] \in \mathfrak{b}(k_1, k_2)\text{-}Lip\langle\delta^\eta, \delta^\rho\rangle$$

▶ **Example 24.** Consider the pseudo-metric space $\langle\wp(\mathbb{Z}), \subseteq, \delta_{siz}\rangle$ and the collecting semantics $[\![\cdot]\!]$. Let the input and output abstractions be $\rho = \eta = \mathsf{Int}$. If we define $+(k_1, k_2) \stackrel{\text{def}}{=} k_1 + k_2$ for any $k_1, k_2 \in \mathbb{R}_{\geq 0}$, then the predicate $\oplus$-Bound$(\langle\delta^{\mathsf{Int}}_{siz}, \delta^{\mathsf{Int}}_{siz}\rangle, +)$ holds. In other words, having

an ALC proof for both P_1 and P_2, with abstract Lipschitz constants k_1, k_2, respectively, gives as result:

$$\begin{aligned}\delta_{siz}((\mathsf{Int}[\![\mathsf{P}_1]\!] \oplus \mathsf{Int}[\![\mathsf{P}_2]\!])c_1, (\mathsf{Int}[\![\mathsf{P}_1]\!] \oplus \mathsf{Int}[\![\mathsf{P}_2]\!])c_2) &\leq k_1\delta_{siz}(\mathsf{Int}(c_1), \mathsf{Int}(c_2)) \\ &\quad + k_2\delta_{siz}(\mathsf{Int}(c_1), \mathsf{Int}(c_2)) \\ &= (k_1 + k_2)\delta_{siz}(\mathsf{Int}(c_1), \mathsf{Int}(c_2))\end{aligned}$$

This is because, when considering δ_{siz} as distance and Int as input and output abstractions, the size of the join of two intervals can be over-approximated by the sum of the number of the elements inside the two intervals. A similar reasoning holds for the quasisemi-metric space $\langle \wp(\mathbb{Z}), \subseteq, \delta_{\subseteq} \rangle$ defined in Ex. 20.

The premise of the rule (**join**) asks for the validity of the following predicates: assume that we have an ALC derivation $k_1 \vdash [\delta^\eta]\,\mathsf{P}_1\,(\delta^\rho)$ for P_1, and $k_2 \vdash [\delta^\eta]\,\mathsf{P}_2\,(\delta^\rho)$ for P_2; if ρ is t-ALC w.r.t. $\langle \delta^{id}, \delta^\rho \rangle$, and the predicate $\oplus$-Bound$(\langle \delta^\eta, \delta^\rho \rangle, \mathfrak{b})$ holds, then we can soundly conclude that the join $\mathsf{P}_1 \oplus \mathsf{P}_2$ is ALC with abstract Lipschitz constant $t\mathfrak{b}(k_1, k_2)$.

Finally, the rule (**star**) deals with loop iterations. It requires that the program P is k-ALC for $\langle \delta^\eta, \delta^\rho \rangle$ and that the input abstraction η is t-ALC for $\langle \delta^\rho, \delta^\eta \rangle$, similar to the (**seq**) rule. In addition, (**star**) requires the assertion $*$-Bound(P^*, m) stating that the loop P^* reaches a least fixpoint in m or less iterations, where m is a constant. This condition can be established either via an auxiliary checker, e.g. an SMT solver, or by manual annotation. Under these premises, we can soundly apply (**star**) in the same way we apply (**seq**), and obtain an abstract Lipschitz constant k^m for the iterations multiplied by the constant t^m generated by applying m-times the abstraction, thus concluding with the $(tk)^m$-ALC of P^*.

The following theorem shows that our proposed deductive system is sound, namely, if $k \vdash [\delta^\eta]\,\mathsf{P}\,(\delta^\rho)$ can be derived by applying the rules of Fig. 4, then $[\![\mathsf{P}]\!]$ satisfies k-ALC w.r.t. $\langle \delta^\eta, \delta^\rho \rangle$. The proof can be found in Appendix B.

▶ **Theorem 25** (Soundness). *Let* $\mathsf{P} \in \mathsf{Prog}$, δ *be a pre-metric and* $\eta, \rho \in uco(C)$ *be the input and output abstractions, respectively. Then:*

$$k \vdash [\delta^\eta]\,\mathsf{P}\,(\delta^\rho) \;\Rightarrow\; [\![\mathsf{P}]\!] \in k\text{-}Lip\langle \delta^\eta, \delta^\rho \rangle$$

▶ **Example 26.** Consider the following program ReLU:

$$(x < 0?\,;\, x := 0) \oplus (x \geq 0?\,;\, \mathbf{skip})$$

implementing the ReLU rectifier function in artificial neural networks [31], that filters the input below 0. Consider the quasisemi-metric space $\langle \wp(\mathbb{Z}), \subseteq, \delta_{\subseteq} \rangle$ and the input and output abstraction $\eta = \rho = \mathsf{Int}$. We want to prove that the collecting semantics $[\![\mathsf{ReLU}]\!] : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ satisfies 1-ALC for $\langle \delta^{\mathsf{Int}}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq} \rangle$. Let us start by analyzing the base commands on the left of $\oplus$. Because the Boolean guard $x < 0?$ is either preserving or removing values from the input, by the rule (**base**), we can derive $1 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\,x < 0?\,(\delta^{\mathsf{Int}}_{\subseteq})$. The command $x := 0$ is neutralizing any distance between input sets since $\delta^{\mathsf{Int}}_{\subseteq}([\![x := 0]\!]S_1, [\![x := 0]\!]S_2) = 0$ for any $S_1, S_2 \in \wp(\mathbb{Z})$. So we can derive $0 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\,x := 0\,(\delta^{\mathsf{Int}}_{\subseteq})$ by the rule (**base**). Since $\mathsf{Int} \in 1\text{-}Lip\langle \delta^{\mathsf{Int}}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq} \rangle$ follows from Lem. 22, we can infer $0 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\,x < 0?\,;\, x := 0\,(\delta^{\mathsf{Int}}_{\subseteq})$ by the rule (**seq**). For the base commands on the right of $\oplus$, we get $1 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\,x \geq 0?\,(\delta^{\mathsf{Int}}_{\subseteq})$ with rule (**base**). The **skip** command does not modify the distance of the input sets, so $1 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\,\mathbf{skip}\,(\delta^{\mathsf{Int}}_{\subseteq})$ can be derived by (**base**). Since $\mathsf{Int} \in 1\text{-}Lip\langle \delta^{\mathsf{Int}}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq} \rangle$ holds, the rule (**seq**) derives $1 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\,x \geq 0?\,;\, \mathbf{skip}\,(\delta^{\mathsf{Int}}_{\subseteq})$. Now for the $\oplus$ operation, we consider $\mathfrak{b}$ as the sum operation $+$ as shown in Ex. 24, thus guaranteeing a sound upper bound for the abstract

Lipschitz constants on the program join. By Lem. 22, the condition $\mathsf{Int} \in 1\text{-}Lip\langle\delta^{id}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq}\rangle$ is equivalent to requiring $\delta_{\subseteq}(\mathsf{Int}(S_1), \mathsf{Int}(S_2)) \leq \delta_{\subseteq}(S_1, S_2)$ for all $S_1, S_2 \in \wp(\mathbb{Z})$, which is clearly satisfied by Int. Therefore, by $\mathsf{Int} \in 1\text{-}Lip\langle\delta^{id}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq}\rangle$, $\oplus\text{-}Bound(\langle\delta^{\mathsf{Int}}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq}\rangle, +)$, $+(0,1) = 1$ and the two derivations on the left and right parts of $\oplus$, we can conclude by the rule (**join**): $1 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\, \mathsf{ReLU}\, (\delta^{\mathsf{Int}}_{\subseteq})$. By Thm. 25, this implies that $[\![\mathsf{ReLU}]\!]$ satisfies 1-ALC for $\langle\delta^{\mathsf{Int}}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq}\rangle$.

Although the proof system of Fig. 4 is sound, it is not complete: there might exist programs that satisfy k-ALC for which the system fails to derive a proof, or for which it only establishes the property with a larger abstract Lipschitz constant $k' \geq k$.

▶ **Example 27.** Consider the program R of Ex. 20 together with the quasisemi-metric space $\langle\wp(\mathbb{Z}), \subseteq, \delta_{\subseteq}\rangle$ and the collecting semantics $[\![\mathsf{R}]\!] : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$. By following similar reasoning done in Ex. 26, we can derive $1 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\, x > 0?\,;\, x := x - 1\, (\delta^{\mathsf{Int}}_{\subseteq})$ and $1 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\, x \leq 0?\,;\, x := x + 1\, (\delta^{\mathsf{Int}}_{\subseteq})$. The rule (**join**) then concludes with $2 \vdash [\delta^{\mathsf{Int}}_{\subseteq}]\, \mathsf{R}\, (\delta^{\mathsf{Int}}_{\subseteq})$ because $+(1,1) = 2$, thus stating that $[\![\mathsf{R}]\!]$ is 2-ALC w.r.t. $\langle\delta^{\mathsf{Int}}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq}\rangle$. Although the conclusion is correct, it is not precise since $[\![\mathsf{R}]\!] \in 1\text{-}Lip\langle\delta^{\mathsf{Int}}_{\subseteq}, \delta^{\mathsf{Int}}_{\subseteq}\rangle$. The imprecision here arises from the bound function $\mathfrak{b} = +$, which overestimates the number of elements produced by the join of two intervals.

For the program R^*, however, the deductive system cannot prove ALC for any constant k. This is due to the fact that rule (**star**) cannot be applied when there is no constant bound m on the number of iterations of R^*, unless the input is restricted to a fixed bound.

As a direct consequence of Thm. 25, if we instantiate the abstractions with $\eta = \rho = id$ and δ is a metric, then the deductive rules of Fig. 4 derive judgments for the standard Lipschitz continuity of programs (cf. Def. 4 with $[\![\mathsf{P}]\!]$ as f). This is because all the predicates on abstractions, such as $\eta \in t\text{-}Lip\langle\delta^{\rho}, \delta^{\eta}\rangle$ for (**seq**) and (**star**), and $\rho \in t\text{-}Lip\langle\delta^{id}, \delta^{\rho}\rangle$ for (**join**), are trivially true (cf. Lem. 22).

▶ **Corollary 28.** *Let* $\mathsf{P} \in \mathsf{Prog}$ *and* δ *be a metric. Then:*

$$k \vdash [\delta^{id}]\, \mathsf{P}\, (\delta^{id}) \;\Rightarrow\; [\![\mathsf{P}]\!] \text{ is } k\text{-Lipschitz continuous w.r.t. } \langle\delta, \delta\rangle$$

▶ **Example 29.** Consider the metric space $\langle\mathbb{M}, \leq, \delta_2\rangle$ where $\leq$ is assumed to be component-wise and δ_2 is the Euclidean distance (cf. Ex 2). Assume that $\mathsf{P} \in \mathsf{Prog}$ is an always terminating program and let $[\![\mathsf{P}]\!] : \mathbb{M} \to \mathbb{M}$ represents the standard denotational semantics mapping a program state to the resulting program state after execution of P. If we instantiate the deductive system of Fig. 4 with the abstraction $\eta = \rho = id$, the semantics $[\![\mathsf{P}]\!] : \mathbb{M} \to \mathbb{M}$ and the metric δ_2, then the inductive rules correspond to those proposed by Chaudhuri et al. in [9]. This shows that the deductive system presented by Chaudhuri et al. in [9] is in fact an instance of $k \vdash [\delta^{\eta}]\, \mathsf{P}\, (\delta^{\rho})$.

6 Related Work

Abstract Lipschitz continuity finds some instances in the literature. Fo instance, 0-ALC corresponds to require: $\forall x, y \in C.\ \delta^{\rho}_{\mathsf{D}}(f(x), f(y)) \leq 0$. When δ_{D} satisfies the (*iff-identity*) axiom, the notion collapses to Abstract Robustness [19] with different models of perturbation (qualitative, quantitative, or combined). In addition, when $\eta = \rho = id$ the notion collapses to the standard program Robustness notion [19].

As we have already discussed in Sec. 4, Partial completeness, whose underlying idea was to replace indistinguishability (of abstract computations) with similarity (measured by a pre-metric distance), has a strong relation with ALC. The same idea in the literature led to another notion that can be seen as an instantiation of our approach, which is Approximate

Non-Interference [34]. This notion, originally introduced in a probabilistic process algebra, requires the *observable* behaviors of two agents under a similarity threshold ε, instead of being identical (as required by standard Non-Interference [21]). Then, we can see Approximate Non-Interference as an instance of ALC, where the observation of the output is the abstraction, and a measured distance between these observables must be under a finite threshold, which is the finite distance between the input processes.

As discussed in Sec. 3, the standard mathematical notion of Lipschitz continuity is an instance of ALC. In particular, when f is the standard input/output denotational program semantics $[\![\cdot]\!]\colon \mathsf{Prog} \to \mathbb{M} \to \mathbb{M}$ and the distance considered is the standard Euclidean metric, then ALC corresponds to the Lipschitz continuity of programs as formalized by Chaudhuri et al. in [9, 10] (referred to as *program Robustness*). We have also shown in Ex. 29 that the proof system in Fig. 4 is a strict generalization of the one in proposed in [9]. This is because the triple $k \vdash [\delta^{\eta}]\,\mathsf{P}\,(\delta^{\rho})$ enables reasoning about property perturbations, encoded with input and output abstractions, over weaker distances (pre-metric spaces) of *any* additive program semantics. It is also worth noting that, in contrast to [9], our proposed deductive system tracks the necessary assumptions for the base cases $[\![\mathsf{c}]\!]$ required to apply the inductive rules, whereas in [9] the authors also provide an analysis for the base cases.

7 Conclusion

Abstract Lipschitz continuity is a generalization of the classical mathematical notion of Lipschitz continuity. It is parameterized by input and output pre-metric spaces, as well as by input and output domain abstractions, which are formalized as upper closure operators. This generalized framework enables the formalization of properties of the form: "*Perturbations in the input properties induce proportionally bounded (linear) changes in the output properties*". We also formally proved its relation with the Partial Completeness property in abstract interpretation, by isolating the constraint under which the two notions, apparently unrelated, have a strong relation. Finally, we developed a deductive system for proving the ALC property of additive semantics of programs.

The proposed ALC notion is a *global* property, in the sense that it is universally quantified over all inputs. As a future work, we plan to formalize its *local* version, namely requiring ALC over a strict subset of the input domain, and study its relation with other local properties in the context of abstract interpretation [2, 3, 5]. Dropping the universal quantification may invalidate the correlation already established between the global counterparts. Also, reasoning about local properties may be more challenging, as the proposed deductive system requires nontrivial modifications to be used for proving ALC on a subset of executions.

We formalized abstractions as ucos, which have been proven to be equivalent to Galois insertions [12], namely admitting a surjective best abstraction function. In the future, we would like to consider weaker abstraction notions able to formalize properties that do not necessarily admit a best abstraction function, such as the domain of convex polyhedra [22]. In this direction, the notion of weak closures defined in [28] could be considered.

Finally, in [28] the authors showed that, under certain assumptions, there is a correspondence between the Completeness property in abstract interpretation and the Abstract Non-Interference (ANI) property in language based security [17, 18]. While ANI does not directly model continuity properties of functions, the connection established in [28], together with Cor. 19, suggests a potential relation between ANI and ALC, which we plan to investigate as future work. The same could also apply to other quantitative program properties, like Quantitative Input Data Usage [29, 30].

References

1. Kumail Alhamoud, Hasan Abed Al Kader Hammoud, Motasem Alfarra, and Bernard Ghanem. Generalizability of adversarial robustness under distribution shifts. *Trans. Mach. Learn. Res.*, 2023, 2023. URL: https://openreview.net/forum?id=XNFo3dQiCJ.
2. Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A logic for locally complete abstract interpretations. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470608.
3. Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A correctness and incorrectness program logic. *J. ACM*, 70(2):15:1–15:45, 2023. doi:10.1145/3582267.
4. Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. Partial (in)completeness in abstract interpretation: limiting the imprecision in program analysis. *Proc. ACM Program. Lang.*, 6(POPL):1–31, 2022. doi:10.1145/3498721.
5. Marco Campion, Mila Dalla Preda, Roberto Giacobazzi, and Caterina Urban. Monotonicity and the precision of program analysis. *Proc. ACM Program. Lang.*, 8(POPL):1629–1662, 2024. doi:10.1145/3632897.
6. Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. On the properties of partial completeness in abstract interpretation. In Ugo Dal Lago and Daniele Gorla, editors, *Proceedings of the 23rd Italian Conference on Theoretical Computer Science, ICTCS 2022, Rome, Italy, September 7-9, 2022*, volume 3284 of *CEUR Workshop Proceedings*, pages 79–85. CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3284/8665.pdf.
7. Marco Campion, Caterina Urban, Mila Dalla Preda, and Roberto Giacobazzi. A formal framework to measure the incompleteness of abstract interpretations. In Manuel V. Hermenegildo and José F. Morales, editors, *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings*, volume 14284 of *Lecture Notes in Computer Science*, pages 114–138. Springer, 2023. doi:10.1007/978-3-031-44245-2_7.
8. Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity Analysis of Programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 57–70, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1706299.1706308.
9. Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity and Robustness of Programs. 55(8):107–115, 2012. doi:10.1145/2240236.2240262.
10. Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara NavidPour. Proving Programs Robust. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 102–112. ACM, 2011. doi:10.1145/2025113.2025131.
11. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.
12. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979. doi:10.1145/567752.567778.
13. Patrick Cousot and Radhia Cousot. An abstract interpretation-based framework for software watermarking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 173–185, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/964001.964016.
14. Lawrence C Evans. *Partial differential equations*, volume 19. American Mathematical Society, 2022.

15 Mahyar Fazlyab, Alexander Robey, Hamed Hassani, Manfred Morari, and George J. Pappas. Efficient and accurate estimation of lipschitz constants for deep neural networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 11423–11434, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/95e1533eb1b20a97777749fb94fdb944-Abstract.html.

16 Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. Analyzing program analyses. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 261–273. ACM, 2015. doi:10.1145/2676726.2676987.

17 Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 186–197. ACM, 2004. doi:10.1145/964001.964017.

18 Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: A unifying framework for weakening information-flow. *ACM Trans. Priv. Secur.*, 21(2):9:1–9:31, 2018. doi:10.1145/3175660.

19 Roberto Giacobazzi, Isabella Mastroeni, and Elia Perantoni. Adversities in abstract interpretation - accommodating robustness by abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 46(2):5, 2024. doi:10.1145/3649309.

20 Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000. doi:10.1145/333979.333989.

21 Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982. doi:10.1109/SP.1982.10014.

22 Branko Grünbaum, Victor Klee, Micha A Perles, and Geoffrey Colin Shephard. *Convex polytopes*, volume 16. Springer, 1967.

23 Yujia Huang, Huan Zhang, Yuanyuan Shi, J. Zico Kolter, and Anima Anandkumar. Training certifiably robust neural networks with efficient local lipschitz bounds. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 22745–22757, 2021. URL: https://proceedings.neurips.cc/paper/2021/hash/c055dcc749c2632fd4dd806301f05ba6-Abstract.html.

24 Ziwei Ji and Matus Telgarsky. Directional convergence and alignment in deep learning. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/c76e4b2fa54f8506719a5c0dc14c2eb9-Abstract.html.

25 Lin Li, Yifei Wang, Chawin Sitawarin, and Michael W. Spratling. Oodrobustbench: a benchmark and large-scale analysis of adversarial robustness under distribution shift. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=kAFevjEYsz.

26 Dennis Liew, Tiago Cogumbreiro, and Julien Lange. Sound and partially-complete static analysis of data-races in GPU programs. *Proc. ACM Program. Lang.*, 8(OOPSLA2):2434–2461, 2024. doi:10.1145/3689797.

27 Francesco Logozzo. Towards a quantitative estimation of abstract interpretations. In *Workshop on Quantitative Analysis of Software*. Microsoft,

737 June 2009. URL: https://www.microsoft.com/en-us/research/publication/
738 towards-a-quantitative-estimation-of-abstract-interpretations/.
739 **28** Isabella Mastroeni and Michele Pasqua. Domain precision in galois connection-less abstract
740 interpretation. In Manuel V. Hermenegildo and José F. Morales, editors, *Static Analysis - 30th*
741 *International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings*,
742 volume 14284 of *Lecture Notes in Computer Science*, pages 434–459. Springer, 2023. doi:
743 10.1007/978-3-031-44245-2_19.
744 **29** Denis Mazzucato, Marco Campion, and Caterina Urban. Quantitative input usage static
745 analysis. In Nathaniel Benz, Divya Gopinath, and Nija Shi, editors, *NASA Formal Methods*
746 *- 16th International Symposium, NFM 2024, Moffett Field, CA, USA, June 4-6, 2024, Pro-*
747 *ceedings*, volume 14627 of *Lecture Notes in Computer Science*, pages 79–98. Springer, 2024.
748 doi:10.1007/978-3-031-60698-4_5.
749 **30** Denis Mazzucato, Marco Campion, and Caterina Urban. Quantitative static timing analysis.
750 In Roberto Giacobazzi and Alessandra Gorla, editors, *Static Analysis - 31st International*
751 *Symposium, SAS 2024, Pasadena, CA, USA, October 20-22, 2024, Proceedings*, volume
752 14995 of *Lecture Notes in Computer Science*, pages 268–299. Springer, 2024. doi:10.1007/
753 978-3-031-74776-2_11.
754 **31** Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann
755 machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th*
756 *International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*,
757 pages 807–814. Omnipress, 2010. URL: https://icml.cc/Conferences/2010/papers/432.
758 pdf.
759 **32** Yurii Nesterov. *Lectures on Convex Optimization*. Springer Publishing Company, Incorporated,
760 2nd edition, 2018. doi:10.1007/978-3-319-91578-4.
761 **33** Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32,
762 2020. doi:10.1145/3371078.
763 **34** Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. *J.*
764 *Comput. Secur.*, 12(1):37–82, 2004. doi:10.3233/JCS-2004-12103.
765 **35** Alessandra Di Pierro and Herbert Wiklicky. Measuring the precision of abstract inter-
766 pretations. In Kung-Kiu Lau, editor, *Logic Based Program Synthesis and Transformation,*
767 *10th International Workshop, LOPSTR 2000 London, UK, July 24-28, 2000, Selected Pa-*
768 *pers*, volume 2042 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2000.
769 doi:10.1007/3-540-45142-0_9.
770 **36** Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation*
771 *perspective*. Mit Press, 2020.
772 **37** Daniel Schoepe and Andrei Sabelfeld. Understanding and enforcing opacity. In *2015 IEEE 28th*
773 *Computer Security Foundations Symposium*, pages 539–553, 2015. doi:10.1109/CSF.2015.41.
774 **38** Pascal Sotin. Quantifying the Precision of Numerical Abstract Domains. Research report,
775 February 2010. URL: https://inria.hal.science/inria-00457324.
776 **39** Bohang Zhang, Du Jiang, Di He, and Liwei Wang. Rethinking lipschitz neural net-
777 works and certified robustness: A boolean function perspective. In Sanmi Koyejo,
778 S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances*
779 *in Neural Information Processing Systems 35: Annual Conference on Neural Informa-*
780 *tion Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 -*
781 *December 9, 2022*, 2022. URL: http://papers.nips.cc/paper_files/paper/2022/hash/
782 7b04ec5f2b89d7f601382c422dfe07af-Abstract-Conference.html.

A Proofs for Section 4 (Abstract Lipschitz Continuity for Programs)

▶ **Theorem 18.** *Let $\langle C, \preceq_{\mathsf{C}}, \delta_{\mathsf{C}}\rangle$ and $\langle D, \preceq_{\mathsf{D}}, \delta_{\mathsf{D}}\rangle$ be pre-metric compatible spaces, let $\eta \in uco(C)$, $\rho \in uco(D)$ be abstractions, and let $k \in \mathbb{R}_{\geq 0}$. Consider a monotone function $f : C \to D$. Then, if f satisfies k-ALC w.r.t. $\langle \delta^{\eta}_{\mathsf{C}}, \delta^{\rho}_{\mathsf{D}}\rangle$, it also satisfies 0-Partial Completeness w.r.t. $\langle \eta, \delta^{\rho}_{\mathsf{D}}\rangle$, namely:*

$$[\forall x, y \in C.\ \delta^{\rho}_{\mathsf{D}}(f(x), f(y)) \leq\ k\delta^{\eta}_{\mathsf{C}}(x, y)]\ \Rightarrow\ [\forall x \in C.\ \delta^{\rho}_{\mathsf{D}}(f(x), f\eta(x)) \leq 0]$$

Proof. Let us assume Abstract k-Lipschitz Continuity w.r.t. $\langle \delta^{\eta}_{\mathsf{C}}, \delta^{\rho}_{\mathsf{D}}\rangle$, namely $\forall x, y \in C.\ \delta^{\rho}_{\mathsf{D}}(f(x), f(y)) \leq\ k\delta^{\eta}_{\mathsf{C}}(x, y)$. We have to prove 0-Partial Completeness w.r.t. $\langle \eta, \delta^{\rho}_{\mathsf{D}}\rangle$, namely $\forall x \in C.\ \delta^{\rho}_{\mathsf{D}}(f(x), f\eta(x)) \leq 0$. Let $y = \eta(x)$. Since Abstract k-Lipschitz Continuity holds, we have $\forall x \in C.\ \delta^{\rho}_{\mathsf{D}}(f(x), f(\eta(x))) \leq\ k\delta^{\eta}_{\mathsf{C}}(x, \eta(x))$ and, by idempotence of η, we have $\delta^{\eta}_{\mathsf{C}}(x, \eta(x)) = 0$. Hence, 0-Partial Completeness w.r.t. $\langle \eta, \delta^{\rho}_{\mathsf{D}}\rangle$ holds. ◀ ◀

▶ **Corollary 19.** *If $\langle D, \preceq_{\mathsf{D}}, \delta_{\mathsf{D}}\rangle$ is a quasisemi-metric compatible space then k-ALC w.r.t. $\langle \delta^{\eta}_{\mathsf{C}}, \delta^{\rho}_{\mathsf{D}}\rangle$ implies Completeness w.r.t. $\langle \eta, \rho\rangle$.*

Proof. Continuing the proof of Thm. 18, we reached 0-Partial Completeness w.r.t. $\langle \eta, \delta^{\rho}_{\mathsf{D}}\rangle$ because, by fixing $y = \eta(x)$ and by the idempotence of η, we get $\forall x \in C.\ \delta^{\rho}_{\mathsf{D}}(f(x), f\eta(x)) \leq 0$. Then, since δ_{D} is a quasisemi-metric, it satisfies the *(iff-identity)* axiom (together with the *(non-negativity)*), and so $\delta^{\rho}_{\mathsf{D}}(f(x), f\eta(x)) \leq 0$ corresponds to $\delta^{\rho}_{\mathsf{D}}(f(x), f\eta(x)) = 0$ which implies $\forall x \in C.\ \rho f(x) = \rho f\eta(x)$. ◀

B Proofs for Section 5 (Proving Abstract Lipschitz Continuity for Programs)

▶ **Theorem 25** (Soundness). *Let $\mathsf{P} \in \mathsf{Prog}$, δ be a pre-metric and $\eta, \rho \in uco(C)$ be the input and output abstractions, respectively. Then:*

$$k \vdash [\delta^{\eta}]\,\mathsf{P}\,(\delta^{\rho})\ \Rightarrow\ [\![\mathsf{P}]\!] \in k\text{-}Lip\langle\delta^{\eta}, \delta^{\rho}\rangle$$

Proof. (**base**): immediate by the definition of $k \vdash [\delta^{\eta}]\,\mathsf{c}\,(\delta^{\rho})$ and the assumption $[\![\mathsf{c}]\!] \in k\text{-}Lip\langle\delta^{\eta}, \delta^{\rho}\rangle$.

(**weaken**): The proof for the weakening of k is immediate. Let us show the proof for weakening the input abstraction η'. Assume $k \vdash [\delta^{\eta'}]\,\mathsf{P}\,(\delta^{\rho})$ and $\eta' \in t\text{-}Lip\langle\delta^{\eta}, \delta^{\eta'}\rangle$. The second assumption can be written as $\forall c_1, c_2 \in C$: $\delta^{\eta'}(\eta'(c_1), \eta'(c_2)) \leq t\delta^{\eta}(c_1, c_2)$ which, by the idempotence property of η', corresponds to $\delta^{\eta'}(c_1, c_2) \leq t\delta^{\eta}(c_1, c_2)$. We get the following derivations $\forall c_1, c_2 \in C$:

$$\begin{aligned}
\delta^{\rho}([\![\mathsf{P}]\!]c_1, [\![\mathsf{P}]\!]c_2)\ &\leq\ [\text{by } k \vdash [\delta^{\eta'}]\,\mathsf{P}\,(\delta^{\rho})]\\
k\delta^{\eta'}(c_1, c_2)\ &\leq\ [\text{by } \eta' \in t\text{-}Lip\langle\delta^{\eta}, \delta^{\eta'}\rangle]\\
tk\delta^{\eta}(c_1, c_2)\ &\Leftrightarrow\ [\text{by judgment definition}]\\
tk \vdash [\delta^{\eta}]\,\mathsf{P}\,(\delta^{\rho})&
\end{aligned}$$

The proof for weakening the output abstraction ρ is similar and therefore omitted.

(**seq**): Assume we have a derivation $k_1 \vdash [\delta^{\eta}]\,\mathsf{P}_1\,(\delta^{\rho})$ for program P_1, a derivation $k_2 \vdash [\delta^{eta}]\,\mathsf{P}_2\,(\delta^{\rho})$ for program P_2, and $\eta \in t\text{-}Lip\langle\delta^{\rho}, \delta^{\eta}\rangle$. By η idempotent, the last assumption can be written as: $\forall c_1, c_2 \in C.\ \delta^{\eta}(c_1, c_2) \leq t\delta^{\rho}(c_1, c_2)$. Then we get the following derivations $\forall c_1, c_2 \in C$:

$$\delta^{\rho}([\![\mathsf{P}_1;\mathsf{P}_2]\!]c_1, [\![\mathsf{P}_1;\mathsf{P}_2]\!]c_2)\ =\ [\text{by definition of } [\![\mathsf{P}_1;\mathsf{P}_2]\!] \text{ and } (\textit{if-identity}) \text{ of } \delta^{\rho}]$$

$$\begin{aligned}
\delta^\rho(\llbracket \mathsf{P}_2 \rrbracket \llbracket \mathsf{P}_1 \rrbracket c_1, \llbracket \mathsf{P}_2 \rrbracket \llbracket \mathsf{P}_1 \rrbracket c_2) &\leq [\text{by } k_2 \vdash [\delta^\eta]\,\mathsf{P}_2\,(\delta^\rho)] \\
k_2\delta^\eta(\llbracket \mathsf{P}_1 \rrbracket c_1, \llbracket \mathsf{P}_1 \rrbracket c_2) &\leq [\text{by } \eta \in t\text{-}Lip\langle \delta^\rho, \delta^\eta \rangle] \\
tk_2\delta^\rho(\llbracket \mathsf{P}_1 \rrbracket c_1, \llbracket \mathsf{P}_1 \rrbracket c_2) &\leq [\text{by } k_1 \vdash [\delta^\eta]\,\mathsf{P}_1\,(\delta^\rho)] \\
k_1 tk_2\delta^\eta(c_1, c_2) &\Leftrightarrow [\text{by judgment definition}] \\
k_1 tk_2 \vdash [\delta^\eta]\,\mathsf{P}_1; \mathsf{P}_2\,(\delta^\rho)
\end{aligned}$$

(**join**): Assume we have a derivation $k_1 \vdash [\delta^\eta]\,\mathsf{P}_1\,(\delta^\rho)$ for program P_1, a derivation $k_2 \vdash [\delta^\eta]\,\mathsf{P}_2\,(\delta^\rho)$ for program P_2, $\rho \in \delta\text{-}Lip\langle t, id\rangle\rho$, and the predicate $\oplus$-Bound$(\langle \eta, \rho\rangle, \mathfrak{b})$ holds for bound function $\mathfrak{b}$. By Lem. 22, the assumption $\rho \in t\text{-}Lip\langle \delta^{id}, \delta^\rho\rangle$ can be written as: $\forall c_1, c_2 \in C.\ \delta^\rho(c_1, c_2) \leq t\delta(c_1, c_2)$. Then we get the following derivations $\forall c_1, c_2 \in C$:

$$\begin{aligned}
\delta^\rho(\llbracket \mathsf{P}_1 \oplus \mathsf{P}_2 \rrbracket c_1, \llbracket \mathsf{P}_1 \oplus \mathsf{P}_2 \rrbracket c_2) &= [\text{by definition of } \llbracket \mathsf{P}_1 \oplus \mathsf{P}_2 \rrbracket \text{ and } (\textit{if-identity}) \text{ of } \delta^\rho] \\
\delta^\rho(\llbracket \mathsf{P}_1 \rrbracket c_1 \vee \llbracket \mathsf{P}_2 \rrbracket c_1, \llbracket \mathsf{P}_1 \rrbracket c_2 \vee \llbracket \mathsf{P}_2 \rrbracket c_2) &= [\text{by } \rho(\rho(c_1) \vee \rho(c_2)) = \rho(c_1 \vee c_2) \text{ and } (\textit{if-identity}) \text{ of } \delta^\rho] \\
\delta^\rho(\rho\llbracket \mathsf{P}_1 \rrbracket c_1 \vee \rho\llbracket \mathsf{P}_2 \rrbracket c_1, \rho\llbracket \mathsf{P}_1 \rrbracket c_2 \vee \rho\llbracket \mathsf{P}_2 \rrbracket c_2) &\leq [\text{by } \rho \in t\text{-}Lip\langle \delta^{id}, \delta^\rho\rangle] \\
t\delta(\rho\llbracket \mathsf{P}_1 \rrbracket c_1 \vee \rho\llbracket \mathsf{P}_2 \rrbracket c_1, \rho\llbracket \mathsf{P}_1 \rrbracket c_2 \vee \rho\llbracket \mathsf{P}_2 \rrbracket c_2) &\leq [\text{by } k_1 \vdash [\delta^\eta]\,\mathsf{P}_1\,(\delta^\rho),\ k_2 \vdash [\delta^\eta]\,\mathsf{P}_2\,(\delta^\rho),\ \oplus\text{-Bound}(\langle \eta, \rho\rangle, \mathfrak{b})] \\
\mathfrak{b}(k_1, k_2)t\delta^\eta(c_1, c_2) &\Leftrightarrow [\text{by judgment definition}] \\
\mathfrak{b}(k_1, k_2)t \vdash [\delta^\eta]\,\mathsf{P}_1 \oplus \mathsf{P}_2\,(\delta^\rho)
\end{aligned}$$

(**star**): Assume we have a derivation $k \vdash [\delta^\eta]\,\mathsf{P}\,(\delta^\rho)$ for program P, $\eta \in t\text{-}Lip\langle \delta^\rho, \delta^\eta\rangle$ and a bound m on the number of iterations by the predicate $*$-Bound(P^*, m). We obtain the following inequalities:

$$\begin{aligned}
\delta^\rho(\llbracket \mathsf{P}^* \rrbracket c_1, \llbracket \mathsf{P}^* \rrbracket c_2) &= [\text{by } *\text{-Bound}(\mathsf{P}^*, m),\ \llbracket \mathsf{P} \rrbracket \text{ additive and } (\textit{if-identity}) \text{ of } \delta^\rho] \\
\delta^\rho(\llbracket \mathsf{P} \rrbracket^m c_1, \llbracket \mathsf{P} \rrbracket^m c_2) &= [\text{by definition of } \llbracket \mathsf{P}; \mathsf{P} \rrbracket \text{ and } (\textit{if-identity}) \text{ of } \delta^\rho] \\
\delta^\rho(\llbracket \mathsf{P} \rrbracket \llbracket \mathsf{P} \rrbracket^{m-1} c_1, \llbracket \mathsf{P} \rrbracket \llbracket \mathsf{P} \rrbracket^{m-1} c_2) &\leq [\text{by } k \vdash [\delta^\eta]\,\mathsf{P}\,(\delta^\rho)] \\
k\delta^\eta(\llbracket \mathsf{P} \rrbracket^{m-1} c_1, \llbracket \mathsf{P} \rrbracket^{m-1} c_2) &\leq [\text{by } \eta \in t\text{-}Lip\langle \delta^\rho, \delta^\eta\rangle] \\
tk\delta^\rho(\llbracket \mathsf{P} \rrbracket^{m-1} c_1, \llbracket \mathsf{P} \rrbracket^{m-1} c_2) &\leq [\text{by applying } m-1 \text{ composition steps}] \\
(tk)^m\delta^\eta(c_1, c_2) &\Leftrightarrow [\text{by judgment definition}] \\
(tk)^m \vdash [\delta^\eta]\,\mathsf{P}^*\,(\delta^\rho)
\end{aligned}$$

◀ ◀

Habilitation to Supervise Research (HDR) Application File

Caterina Urban

Research Project

Projet de recherche

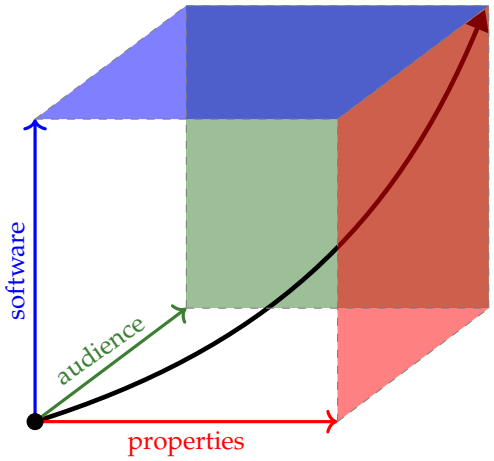
6

Static Analyses for the Properties, Programs, and People of Tomorrow

Analyses statiques pour les propriétés, les programmes et le public de demain

The increasing complexity of modern software systems makes ensuring their correctness, security, and trustworthiness more challenging than ever. As software grows larger and more interconnected, static program analysis, in my view, is the only approach with the potential to keep pace — provided we significantly push beyond its current boundaries. To meet the demands of modern software, we need static analyses that can adapt to heterogeneous software that interacts and evolves over time.

I aim to address these challenges by providing innovative static analysis methods and tools for the *software properties*, *software programs*, and the *software users* of tomorrow.



Specifically, I structure my research program around the following three interconnected research axes:

Practical and Adaptive Static Analyses: The first axis tackles the design and implementation of practical static analyses for *complex properties across multiple software programs*. It also investigates the formal foundations and development of *just-in-time static analyzers* that can self-adapt and adapt the analyzed programs – tailoring precision, performance, and abstractions to the intensional or extensional properties of the program under analysis.

Verification and Explanation of Machine Learning Software: The second research axis is specifically dedicated to machine learning

software. Its goal is to develop *incremental* and *compositional* methods for verification and explainability, by leveraging the internal representation of machine learning models.

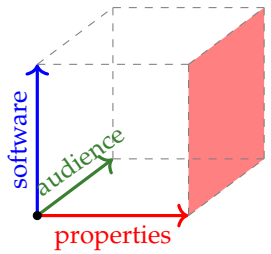
Modern Software Systems: The third axis lifts the challenges addressed by the previous axes from individual programs to the larger context in which they are deployed – distributed systems and pipelines of continuously and asynchronously evolving software components.

6.1 Practical and Adaptive Static Analyses

There is a growing and increasingly diverse landscape of interesting program properties that have been explored from a theoretical perspective over the years: *(partial) abstract non-interference* [Giacobazzi04, u3], *program diversity* [Pucella06], *(abstract) Lipschitz continuity* [Chaudhuri10, u2], *responsibility* [Deng19]. However, there remains a remarkable lack of static analyses capable of handling these properties effectively in practice for realistic software programs. Notable few exceptions are static analyses for *dependency fairness* [c17] – an instance of abstract non-interference – and for program *portability* [Delmas21] – which can be seen as an instance of program diversity. Rather than merely handling specific instances, we need static analyses providing a more general support for these properties. I am particularly interested in developing more general static analyses for reasoning about input *data usage* [c12] and *non-exploitability* [Parolini24, u4], as well as their "higher-order" combination, e.g., determining whether an external adversary can force some input to become (ir)relevant to the behavior of a program. More broadly, I aim to develop static analyses for *non-exploitability of general program hyperproperties*, and general *responsibility* [Deng19] *static analyses for hyperproperties*, i.e., determining the responsible entity for a certain hyperproperty of interest.

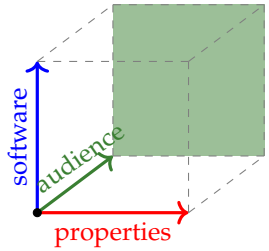
At the same time, I would like to deepen the study of the relationship between *intensional* [Giacobazzi15] and *extensional* [c24, u1] program properties and the *precision* of a static program analysis [c22]. The aim is to draw practical insights to develop more effective static analyses. It is well known that syntactic transformations that preserve the semantics of a program – such as code obfuscation [Collberg09] – can be explicitly designed to degrade the precision of program analysis [DallaPreda05]. Vice versa, I would like to formally study semantic-preserving program transformations that *increase* static analysis precision. In particular, leveraging the general partial completeness framework [c22] for comparing static analyses, I aim to formally characterize the *potency* [Giacobazzi12] of program transformations in relation to the precision of static analysis *across different programs*. I also would like to define a formal framework for the intensional and extensional *sensitivity* of an abstract domain, in terms of the variation in precision of the static analysis employing the abstract domain *across* behavior-preserving *transformations of the same program*. Building on these foundations and drawing inspiration from JIT-compilers, I envision the design of *static analyzers that perform JIT-style transformations* of (fragments of) programs at analysis time – not to improve runtime performance, but to improve analysis precision. A combination with a dynamic analysis that observes concrete program executions would enable speculative program transformations. It will then be necessary to design appropriate backtracking and recovery mechanisms, to handle unsuccessful speculation.

[u1] Campion et al. - Kernel Properties in Abstract Interpretation
[u2] Campion et al. - Abstract Lipschitz Continuity
[u3] Campion et al. - Measuring vs Abstracting: On the Relation between Distances and Abstract Domains
[u4] Moussaoui Remil and Urban - Termination Resilience Static Analysis



[c22] Campion et al. - A Formal Framework to Measure the Incompleteness of Abstract Interpretations (SAS 2023)
[c24] Campion et al. - Monotonicity and the Precision of Program Analysis (POPL 2024)
[Chaudhuri10] Chaudhuri et al. - Continuity Analysis of Programs (POPL 2010)
[Collberg09] Collberg and Nagra - Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection (2009)
[DallaPreda05] Dalla Preda and Giacobazzi - Control Code Obfuscation by Abstract Interpretation (SEFM 2005)
[Delmas21] Delmas et al. - Static Analysis of Endian Portability by Abstract Interpretation (SAS 2021)
[Deng19] Deng and Cousot - Responsibility Analysis by Abstract Interpretation (SAS 2019)
[Giacobazzi04] Giacobazzi and Mastroeni - Abstract Non-Inteference: Parametrizing Non-Interference by Abstract Interpretation (POPL 2004)
[Giacobazzi12] Giacobazzi and Mastroeni - Making Abstract Interpretation Incomplete: Modeling the Potency of Obfuscation (SAS 2012)
[Giacobazzi15] Giacobazzi et al. - Analyzing Program Analyses (POPL 2015)
[Parolini24] Parolini and Miné - Sound Abstract Nonexploitability Analysis (VMCAI 2024)
[Pucella06] Pucella and Schneider - Independence from Obfuscation: A Semantic Framework for Diversity (CSFW 2006)
[c12] Urban and Müller - An Abstract Interpretation Framework for Input Data Usage (ESOP 2018)
[c17] Urban et al. - Perfectly Parallel Fairness Certification of Neural Networks (OOPSLA 2020)

6.2 Verification and Explanation of Machine Learning Software



Recent years have seen the flourishing of formal methods for machine learning [Urban21]. However, the limited scalability of current verification and explainability methods remains a major challenge, hindering further progress in this area. Numerous tools have been developed that achieve remarkable performance by specializing in the verification of local robustness within narrow neighborhoods around reference points – most often insufficient for providing any guarantees beyond the reference points themselves [Geng23]. Scalability of verification methods for more meaningful properties is nowadays limited to models with less than thousands parameters [c17]. Formal logic-based explainability methods face similar scalability barriers [Izza24, Wu23].

I believe that *verifying and explaining large, complex, state-of-the-art machine learning models can be achieved through proper abstractions built on top of their internal representations*. Specifically, I aim to design *specification predicates* offering a compact and expressive vocabulary to characterize how a given machine learning model internally processes information. These predicates, defined at various level of granularity – from activation patterns over individual neurons [c17, Geng23] to higher-level architectural components, enable a shift from reasoning over low-level inputs to reasoning over the structure and dynamics of the model itself. Efficient algorithms to extract these predicates from a given machine learning model will then need to be designed. In particular, I aim to investigate complementary *bottom-up* (from the most concrete specification predicate) *and top-down* (from the most abstract predicate) *search strategies* with iterative generalization or refinements steps. The learned predicates will serve as general-purpose specifications of model behavior for downstream verification and explainability methods. For third-party or otherwise closed-source machine learning models, which are not amenable to static analysis, I envision leveraging concrete executions to speculatively construct specification predicates.

Developing verification methods tailored to work with specification predicates alongside (or even replacing) raw input specification allows for a variety of scalability-enabling strategies. First of all, specification predicates (e.g., based on neural activation patterns [c17]) can be leveraged to prune irrelevant regions of the search space during verification. They also enable the design of *incremental verification approaches* [Ugare23] to reuse intermediate results of similar verification runs – those matching the same specification predicate – and thus avoid redundant computations. Likewise, *compositional verification* becomes feasible by leveraging the modularity of model architectures: individual layers, attention heads, or specialized sub-models can be verified in isolation with respect to their local specification predicates, and their guarantees then composed to establish properties for the whole model under verification.

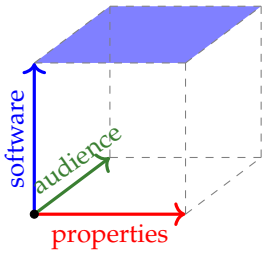
I am also interested in using specification predicates as the foundation for model explanations. Rather than merely highlighting which input features are relevant to a model prediction [Marques-Silva24], explanations grounded in internal model representation can reveal how *relationships*

[Geng23] Geng et al. - Towards Reliable Neural Specifications (ICML 2023)
[Marques-Silva24] Marques-Silva - Logic-Based Explainability: Past, Present and Future (ISoLA 2024)
[Izza24] Izza et al. - Distance-Restricted Explanations: Theoretical Underpinnings & Efficient Implementation (KR 2024)
[Ugare23] Ugare et al. - Incremental Verification of Neural Networks (PLDI 2023)
[c17] Urban et al. - Perfectly Parallel Fairness Certification of Neural Networks (OOPSLA 2020)
[Urban21] Urban and Miné - A Review of Formal Methods applied to Machine Learning (2021)
[Wu23] Wu et al. - VeriX: Towards Verified Explainability of Deep Neural Networks (NeurIPS 2023)

For example, rather than merely indicating that the input feature "age" is important in predicting health outcomes, explanations may instead reveal how "age" interacts with lifestyle choices and medical history of an individual.

between features influence model behavior. These explanations provide a richer narrative that makes model reasoning more accessible. Understanding the relationships between features can also provide insights into potential biases or unintended correlations in a model.

6.3 Modern Software Systems



[Delmas19] Delmas and Miné - Analysis of Software Patches Using Numerical Abstract Interpretation (SAS 2019)
[Delmas21] Delmas et al. - Static Analysis of Endian Portability by Abstract Interpretation (SAS 2021)
[Zanatta24] Zanatta et al. - Sound Static Analysis for Microservices: Utopia? A Preliminary Experience with LiSA (FTfJP@ECOOP 2024)

Modern software is rarely monolithic. Instead, it comprises an interconnected constellation of independently developed components, such as distributed *microservice architectures* or *machine learning pipelines* with each stage – data ingestion, feature engineering, model training, evaluation – handled at different times and in separate environments (different tools, sometimes even separate development teams and organizations). Crucially, each software component in these systems may follow its own development lifecycle and rapidly evolve over time. Changes in one component can (silently) propagate to others, affecting the overall system.

Reasoning about such modern systems requires static analysis techniques that go beyond verifying isolated software components and account for the behavior that emerges from their complex interplay [Zanatta24] – a challenge that calls for *modular yet relational static analyses* capable of reasoning across component boundaries. I aim to leverage the static analyses developed within the previous research axes to shed light on the *observable behavior* of individual software components in a system – notably the static analyses for the hyperproperties of data usage, non-exploitability, responsibility developed within the first research axis (cf. Section 6.1) and, for machine learning components, the specification predicates and explanation methods designed within the second research axis (cf. Section 6.2). By building on the exposed observable behavior in a principled way, I then aim to design and develop static analyses that can reason about properties of the ecosystem in which software components are deployed.

The analyses from Section 6.1 characterize what a component exposes to the rest of the system, e.g., (un)used data or vulnerable variables. The explainability methods from Section 6.2 yield interpretable contracts for machine learning components. These serve as lightweight behavioral interfaces between software components, suitable for reasoning compositionally about their interactions.

I am particularly interested in static analyses that aid understanding how systems behave *across different versions* of their software components [Delmas19, Delmas21], and what properties emerge – at the system level – through that evolution. I envision static analyses that expose how version changes affect the behavior and hyperproperties of software components and, more importantly, how they reflect on (the behavior and hyperproperties of) other software components within a system. Ultimately, this enable static analyses that can reasons about trajectories of evolving software systems, rather than isolated snapshots, e.g., identifying any regression on system-level properties introduced by updated software components.

For example, static analyses of machine learning pipelines that detect whether a version change of a data preprocessing component invalidates assumptions made downstream by a model, or causes a regression in fairness guarantees between retrainings of a classifiers.

These questions are further complicated in the presence of *trust boundaries*: while some software components are in-house and trusted, others may be third-party, or even automatically generated. In the context of machine learning, trust extends to datasets and models. I aim to develop static analyses that establish system-level guarantees in spite of untrusted software components – or even when the static analysis of untrusted components may be itself untrusted (e.g., speculative). To this end, I plan on leveraging static analyses for non-exploitability of hyperproperties developed in the first research axis, to verify that adversarial behavior of an untrusted component cannot affect the behavior of other component, or compromise global properties of the whole system.

Habilitation to Supervise Research (HDR) Application File

Caterina Urban

Doctoral Thesis Report

Rapport de thèse de doctorat

ENS
ÉCOLE NORMALE SUPÉRIEURE

RAPPORT DE SOUTENANCE
Procès verbal
Thèse de doctorat

NOM et prénom : URBAN Caterina Année de soutenance / n° étudiant : 2015 / n° 190082
Née le : 9 mars 1987 à Udine (Italie)

Date de soutenance : 9 juillet 2015
Sujet de thèse: « *Analyse statique par interprétation abstraite de propriétés temporelles fonctionnelles des programmes* »
École doctorale : Sciences Mathématique de Paris Centre (ED N° 386)
Spécialité : Informatique

Président du jury (*) : Nicolas Halbwachs

☒ En cas de visioconférence le président t du jury atteste de la bonne transmission des échanges durant toute la durée de la soutenance

Prénom NOM	Qualité	Établissement d'exercice	Signature
Antoine MINE	Chargé de recherche	ENS	
Manuel HERMENEGILDO	Full Professeur	IMDEA Software Institut	par visio
Andreas PODELSKI	Professeur	University of Freiburg	
Nicolas HALBWACHS	Directeur de recherche	IMAG	
Patrick COUSOT	Professeur	ENS	
Mooly SAGIV	Professeur	Tel Aviv University	par visio

Mention délivrée :
☐ Pas de mention
☐ Honorable
☐ Très Honorable
☒ Très Honorable avec les félicitations du jury à l'unanimité (**)

Fait à Paris, le 9 juillet 2015
Signature du Président du jury,

*(*) **Arrêté ministériel du 7 août 2006 - article 19** : Les membres du jury désignent parmi eux un président et le cas échéant, un rapporteur de soutenance. Le président du jury doit être un professeur ou assimilé ou un enseignant de rang équivalent qui ne dépend pas du ministère chargé de l'enseignement supérieur. Le directeur de thèse, s'il participe au jury, ne peut être choisi ni comme rapporteur de soutenance, ni comme président du jury.*

*(**) **Arrêté ministériel du 7 août 2006 - article 20** : La plus haute mention, qui est réservée à des candidats aux qualités exceptionnelles démontrées par les travaux et la soutenance, ne peut être décernée qu'après un vote à bulletin secret et unanime des membres du jury. Dans ce cas, le président du jury établit un rapport complémentaire justifiant cette distinction.*

ENS
ÉCOLE NORMALE SUPÉRIEURE

RAPPORT DE SOUTENANCE
Thèse de doctorat
de Mme Caterina URBAN
École doctorale N° 386

Date de soutenance : 9 juillet 2015
Sujet de thèse: « *Analyse statique par interprétation abstraite de propriétés temporelles fonctionnelles des programmes* »

Rapport suivi de la signature du Président du jury :

.....

Caterina Urban a présenté son travail sur l'analyse par interprétation abstraite de propriétés de terminaison et de vivacité des programmes. Elle en a fait un excellent exposé, situé au bon niveau d'abstraction, et fournissant des explications intuitives sur des algorithmes absolument non triviaux. La qualité des planches a également été appréciée.

Aux nombreuses questions du jury, elle a fourni des réponses précises et enthousiastes, montrant sa grande expertise d'un domaine dépassant le strict sujet de la thèse. Elle a également esquissé de manière pertinente les prolongements possibles de son travail, et ses perspectives de recherche, montrant ainsi sa maturité scientifique et son autonomie.

Le jury tient à souligner le niveau exceptionnel du travail effectué, sur un sujet difficile. Il a noté que ce travail est le premier à démontrer l'applicabilité pratique de l'interprétation abstraite à la preuve de propriétés de vivacité.

En conséquence, le jury a décerné à Caterina Urban le grade de Docteur de l'Ecole Normale Supérieure, avec la mention "très honorable" et les félicitations du jury.

.....
.....
.....
.....
.....
.....
.....
.....
.....

Fait à Paris, le 9.07.2015
Signature du Président du jury,

ENS
ÉCOLE NORMALE
SUPÉRIEURE

Justification des félicitations décernées à Caterina Urban par le jury de sa thèse

Le travail de thèse de Caterina Urban a été jugé exceptionnel par tous les membres du jury, par l'originalité et la profondeur de la contribution scientifique à un sujet tres difficile, concurrentiel, et d'une importance pratique évidente.

Les rapporteurs ont souligné les apports novateurs contenus dans la thèse, en particulier une application complète de l'interprétation abstraite à la vérification de propriétés de vivacité, le calcul de préconditions suffisantes de terminaison, et une méthode nouvelle pour inférer des arguments de terminaison sur des traces infinies. Cette thèse et les excellentes publications qui l'accompagnent feront date dans le domaine de l'analyse des programmes, notamment concernant les propriétés de vivacité.

A cette contribution théorique et algorithmique impressionante, Caterina Urban a joint une implémentation complète et solide (disponible en ligne), ainsi qu'une expérimentation convaincante, montrant que son prototype est compétitif avec des outils développés sur de longues périodes par des équipes entières.

Le document est à la fois rigoureux et très pédagogique, malgré la difficulté du contenu. La qualité de la soutenance a été saluée, tant du point de vue de l'exposé que de la réponse aux questions.

C'est pourquoi les membres du jury ont été unanimes à proposer que lui soient décernées les félicitations.

Fait à Paris, le 3-07-2015
Signature du Président du jury,

ENS
ÉCOLE NORMALE SUPÉRIEURE

AVIS DU JURY
Thèse de doctorat
de Mme Caterina URBAN
École doctorale N° 386

NOM et prénom : URBAN Caterina

Date de soutenance : 9 juillet 2015
Sujet de thèse: « *Analyse statique par interprétation abstraite de propriétés temporelles fonctionnelles des programmes* »

Avis sur les corrections

☒ Thèse pouvant être reproduite en l'état

☐ Thèse pouvant être reproduite après corrections mineures sous la seule responsabilité du doctorant (délai : 1 mois)

☐ Thèse pouvant être reproduite après corrections majeures (délai : 3 mois)
Membre du jury désigné pour la vérification des corrections majeures :

.....

Clause de confidentialité décidée par le directeur de l'ENS

☐ oui

☒ non

Date :

Signature du Président du jury :

Les corrections attendues ont été effectuées :

Date :

Signature du Président du jury :

ÉCOLE NORMALE SUPÉRIEURE
45 rue d'Ulm - 75230 Paris Cedex 05 - FRANCE
Tél. : +33 (0)1 44 32 30 00 - www.ens.fr