A Guided Tour of a Static Analyzer for Data Science Software

Caterina Urban
ANTIQUE Research Team
INRIA & École Normale Supérieure, Paris, France
Can AI Be a Fair Judge in Court? Estonia Thinks So

Estonia plans to use an artificial intelligence program to decide some small-claims cases, part of a push to make government services smarter.

In 2019, predictive algorithms will start to make banking fair for all

In a nursing home where his mother visited him daily, waiting until a devastating fall in 2015. He had spent time in the hospital, and by 2016

4 ways to check for skin cancer with your smartphone

Your phone can help you recognize suspicious moles and marks, but you should still see a dermatologist or doctor.

Deep Neural Network Compression for Aircraft Collision Avoidance Systems

Abstract—One approach to designing decision making logic for an aircraft collision avoidance system frames the problem as a partially observable Markov decision process and optimizes the system using dynamic programming. The resulting collision avoidance strategy can be represented as a numeric table. This methodology has been used in the development of the Airborne Collision Avoidance System X (ACAS X) family of collision avoidance systems for manned and unmanned aircraft, but the high dimensionality of the state space leads to very large tables. To improve storage efficiency, a deep neural network is used to approximate the table. With the use of an asymmetric loss function and a gradient descent algorithm, the parameters for this network can be trained to provide accurate estimates of table values while preserving the relative preferences of the possible advisories for each state. By training multiple networks to represent suitable tables, the network also decreases the required run-time for computing the collision avoidance advisory. Simulation studies show that the network improves the safety and efficiency of the collision avoidance system. Because only the network parameters need to be stored, the required storage space is reduced by a factor of 1000, enabling the collision avoidance system to operate using current avionics systems.

1. INTRODUCTION

Decades of research have explored a variety of approaches to designing decision making logic for aircraft collision avoidance systems for both manned and unmanned aircraft [1]. Recent work on formulating the problem of collision avoidance as a partially observable Markov decision process (POMDP) has led to the development of the Airborne Collision Avoidance System X (ACAS X) family of collision avoidance systems [2], [3], [4]. The version for manned aircraft, ACAS Xa, is expected to become the next international standard for unmanned aircraft, ACAS Xu, uses dynamic programming to solve the decision making logic results in a large numeric table that contains scores associated with different actions as per the ACAS Xa tables that limit advisories to vertical maneuvers, but the ACAS Xu tables for horizontal maneuvers are much larger. Recent work explored a new algorithm that exploits the score table's natural symmetry to remove redundancy within the table [9]. However, results showed that this compression algorithm could not achieve sufficient reduction in storage before compromising performance.

Automated background checks are deciding who’s fit for a home

AI used for first time in job interviews in UK to find best applicants

By Charles Hymas

27 SEPTEMBER 2019 • 10:00 PM

Use of our site with our social media, advertising and analytics partners. We also share information about your community and to analyse our traffic. We also share information about your features and to analyse our traffic. We also share information about your features and to analyse our traffic.
Machine Bias

There's software used across the country to predict future criminals. And it's biased against blacks.
by Julia Angwin, Jeff Larson, Surya Mattu and Lauren Kirchner, ProPublica
May 23, 2016

IBM's Watson supercomputer recommended 'unsafe and incorrect' cancer treatments, internal documents show

By Casey Ross and Ike Swetlitz
July 25, 2018

How Target Figured Out A Teen Girl Was Pregnant Before Her Father Did

Kashmir Hill Former Staff
Tech
Welcome to The Not-So-Private Parts where technology & privacy collide

Machine Bias

There's software used across the country to predict future criminals. And it's biased against blacks.
by Julia Angwin, Jeff Larson, Surya Mattu and Lauren Kirchner, ProPublica
May 23, 2016

Amazon scraps secret AI recruiting tool that showed bias against women

Jeffrey Dastin

How Target Figured Out A Teen Girl Was Pregnant Before Her Father Did

Kashmir Hill Former Staff
Tech
Welcome to The Not-So-Private Parts where technology & privacy collide

 Millions of black people affected by racial bias in health-care algorithms

Study reveals rampant racism in decision-making software used by US hospitals — and highlights ways to correct it.
24 October 2019
Heidi Ledford

A self-driving Uber ran a red light last December, contrary to company claims

Internal documents reveal that the car was at fault

By Andrew Liptak | @AndrewLiptak | Feb 25, 2017, 11:08am EST
Data Science Pipeline

- pre-processing
- training
- data analysis

MACHINE LEARNING

DATA SCIENCE
Data is Dirty

- pre-processing
- training
- data analysis

- inconsistent data
- incorrect data
- incomplete data
- inaccurate data
Pre-Processing is Fragile

For Big-Data Scientists, ‘Janitor Work’ Is Key Hurdle to Insights

By Steve Lohr

Aug. 17, 2014

Technology revolutions come in measured, sometimes foot-dragging steps. The lab science and marketing enthusiasm tend to underestimate the bottlenecks to progress that must be overcome with hard work and practical engineering.

The field known as “big data” offers a contemporary case study. The catchphrase stands for the modern abundance of digital data from many sources — the web, sensors, smartphones and corporate databases — that can be mined with clever software for discoveries and insights. Its promise is smarter, data-driven decision-making in every field. That is why data scientist is the economy’s hot new job.

Yet far too much handcrafted work — what data scientists call “data wrangling,” “data munging” and “data janitor work” — is still required. Data scientists, according to interviews and expert estimates, spend from 50 percent to 80 percent of their time mired in this more mundane labor of collecting and preparing unruly digital data, before it can be explored for useful nuggets.

“Data wrangling is a huge — and surprisingly so — part of the job,” said Monica Rogati, vice president for data science at Jawbone, whose sensor-filled wristband and software track activity, sleep and food consumption, and suggest dietary and health tips based on the numbers. “It’s something that is not appreciated by data civilians. At times, it feels like everything we do.”

Several start-ups are trying to break through these big data bottlenecks by developing software to automate the gathering, cleaning and organizing of disparate data, which is plentiful but messy. The modern Wild West of data needs to be tamed somewhat so it can be recognized and exploited by a computer program.

“It’s an absolute myth that you can send an algorithm over raw data and have insights pop up,” said Jeffrey Heer, a professor of computer science at the University of Washington and a co-founder of Trifacta, a start-up based in San Francisco.

Technology revolutions come in measured, sometimes foot-dragging steps. The lab science and marketing enthusiasm tend to underestimate the bottlenecks to progress that must be overcome with hard work and practical engineering.

The field known as “big data” offers a contemporary case study. The catchphrase stands for the modern abundance of digital data from many sources — the web, sensors, smartphones and corporate databases — that can be mined with clever software for discoveries and insights. Its promise is smarter, data-driven decision-making in every field. That is why data scientist is the economy’s hot new job.

Yet far too much handcrafted work — what data scientists call “data wrangling,” “data munging” and “data janitor work” — is still required. Data scientists, according to interviews and expert estimates, spend from 50 percent to 80 percent of their time mired in this more mundane labor of collecting and preparing unruly digital data, before it can be explored for useful nuggets.

“Data wrangling is a huge — and surprisingly so — part of the job,” said Monica Rogati, vice president for data science at Jawbone, whose sensor-filled wristband and software track activity, sleep and food consumption, and suggest dietary and health tips based on the numbers. “It’s something that is not appreciated by data civilians. At times, it feels like everything we do.”

Several start-ups are trying to break through these big data bottlenecks by developing software to automate the gathering, cleaning and organizing of disparate data, which is plentiful but messy. The modern Wild West of data needs to be tamed somewhat so it can be recognized and exploited by a computer program.

“It’s an absolute myth that you can send an algorithm over raw data and have insights pop up,” said Jeffrey Heer, a professor of computer science at the University of Washington and a co-founder of Trifacta, a start-up based in San Francisco.

For Big-Data Scientists, ‘Janitor Work’ Is Key Hurdle to Insights

By Steve Lohr

Aug. 17, 2014

Technology revolutions come in measured, sometimes foot-dragging steps. The lab science and marketing enthusiasm tend to underestimate the bottlenecks to progress that must be overcome with hard work and practical engineering.

The field known as “big data” offers a contemporary case study. The catchphrase stands for the modern abundance of digital data from many sources — the web, sensors, smartphones and corporate databases — that can be mined with clever software for discoveries and insights. Its promise is smarter, data-driven decision-making in every field. That is why data scientist is the economy’s hot new job.

Yet far too much handcrafted work — what data scientists call “data wrangling,” “data munging” and “data janitor work” — is still required. Data scientists, according to interviews and expert estimates, spend from 50 percent to 80 percent of their time mired in this more mundane labor of collecting and preparing unruly digital data, before it can be explored for useful nuggets.

“Data wrangling is a huge — and surprisingly so — part of the job,” said Monica Rogati, vice president for data science at Jawbone, whose sensor-filled wristband and software track activity, sleep and food consumption, and suggest dietary and health tips based on the numbers. “It’s something that is not appreciated by data civilians. At times, it feels like everything we do.”

Several start-ups are trying to break through these big data bottlenecks by developing software to automate the gathering, cleaning and organizing of disparate data, which is plentiful but messy. The modern Wild West of data needs to be tamed somewhat so it can be recognized and exploited by a computer program.

“It’s an absolute myth that you can send an algorithm over raw data and have insights pop up,” said Jeffrey Heer, a professor of computer science at the University of Washington and a co-founder of Trifacta, a start-up based in San Francisco.
Accuracy is Meaningless

*Geoffrey Hinton*  
@geoffreyhinton

Suppose you have cancer and you have to choose between a black box AI surgeon that cannot explain how it works but has a 90% cure rate and a human surgeon with an 80% cure rate. Do you want the AI surgeon to be illegal?
The Dark Secret at the Heart of AI

No one really knows how the most advanced algorithms do what they do. That could be a problem.

by Will Knight

Apr 11, 2017
Software = Problems
The Lyra Static Analyzer
The Lyra Static Analyzer
Python
Most Popular Programming Language for Data Science

[Bar chart showing popularity of Python vs. other languages]

The Lyra Static Analyzer

Python is a dynamically-typed language
Typpete
SMT-based Static Type Inference for Python 3.x

def main(a):
    b = a[0] == 0
    ...

def main(a: List[int]) -> int:
    ...

Type error: argument in line 3

subtype(bool, b)
...

Sat. Model:
a = list(int), b = bool, ...

Unsat

https://github.com/caterinaurban/Typpete
Typpete
SMT-based Static Type Inference for Python 3.x

MaxSMT-Based Type Inference for Python 3

Mostafa Hassan1,2, Caterina Urban2, Marco Eilers2, and Peter M¨uller2
1 German University in Cairo, Cairo, Egypt
2 Department of Computer Science, ETH Zurich, Zurich, Switzerland
caterina.urban@inf.ethz.ch

Abstract. We present Typpete, a sound type inference tool that automatically infers Python 3 type annotations. Typpete encodes type constraints as a MaxSMT problem and uses optional constraints and specific quantifier instantiation patterns to make the constraint solving process efficient. Our experimental evaluation shows that Typpete scales to real world Python programs and outperforms state-of-the-art tools.

1 Introduction
Dynamically-typed languages like Python have become increasingly popular in the past five years. Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing offers early error detection, efficient execution, and machine-checked code documentation, and enables more advanced static analysis and verification approaches [15]. For these reasons, Python’s PEP484 [25] has recently introduced optional type annotations in the spirit of gradual typing [23]. The annotations can be checked using MyPy [10]. In this paper, we present our tool Typpete, which automatically infers sound (non-gradual) type annotations and can therefore serve as a preprocessor for other analysis or verification tools.

Typpete performs whole-program type inference, as there are no principal typings in object-oriented languages like Python [1], example in Sect. 1; the inferred types are correct in the given context but may not be as general as possible. The type inference is constraint-based and relies on the off-the-shelf SMT solver Z3 [7] for finding a valid type assignment for the input program. We show that two main ingredients allow Typpete to scale to real programs: (1) a careful encoding of subtyping that leverages efficient quantifier instantiation techniques [6], and (2) the use of optional type equality constraints, which considerably reduce the solution search space. Whenever a valid type assignment cannot be found, Typpete encodes type error localization as an optimization problem [19] and reports only a minimal set of unfulfilled constraints to help the user pinpoint the cause of the error.
Abstract.
We present Typpete, as our type inference tool that automatically infers Python 3 type annotations.
Typpete encodes type constraints as a MaxSMT problem and uses optional constraints and specific quantifier instantiation patterns to make the constraint solving process efficient. Our experimental evaluation shows that Typpete scales to real world Python programs and outperforms state-of-the-art tools.

1 Introduction
Dynamically-typed languages like Python have become increasingly popular in the past five years. Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing offers early error detection, efficient execution, and machine-checked code documentation, and enables more advanced static analysis and verification approaches [15]. For these reasons, Python's PEP484 [25] has recently introduced optional type annotations in the spirit of gradual typing [23]. The annotations can be checked using MyPy [10]. In this paper, we present our tool Typpete, which automatically infers sound (non-gradual) type annotations and can therefore serve as a preprocessor for other analysis or verification tools.

Typpete performs whole-program type inference, as there are no principal typings in object-oriented languages like Python [1]; the inferred types are correct in the given context but may not be as general as possible. The type inference is constraint-based and relies on the off-the-shelf SMT solver Z3 [7] for finding a valid type assignment for the input program. We show that two main ingredients allow Typpete to scale to real programs: (1) a careful encoding of subtyping that leverages efficient quantifier instantiation techniques [6], and (2) the use of optional type equality constraints, which considerably reduce the solution search space. Whenever a valid type assignment for the input program cannot be found, Typpete encodes type error localization as an optimization problem [19] and reports only minimal to fulfill constraints to help the user pinpoint the cause of the error.
Abstract.

We present Typpete, as our type inference tool that automatically infers Python 3 type annotations. Typpete encodes type constraints as a MaxSMT problem and uses optional constraints and specific quantifier instantiation patterns to make the constraint solving process efficient. Our experimental evaluation shows that Typpete scales to real world Python programs and outperforms state-of-the-art tools.

1 Introduction

Dynamically-typed languages like Python have become increasingly popular in the past five years. Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing offers early error detection, efficient execution, and machine-checked code documentation, and enables more advanced static analysis and verification approaches [15].

For these reasons, Python's PEP484 [25] has recently introduced optional type annotations in the spirit of gradual typing [23]. The annotations can be checked using MyPy [10]. In this paper, we present our tool Typpete, which automatically infers sound (non-gradual) type annotations and can therefore serve as a preprocessor for other analysis or verification tools.

Typpete performs whole-program type inference, as there are no principal typings in object-oriented languages like Python [1, example in Sect. 1]; the inferred types are correct in the given context but may not be as general as possible. The type inference is constraint-based and relies on the off-the-shelf SMT solver Z3 [7] for finding a valid type assignment for the input program.

We show that two main ingredients allow Typpete to scale to real programs: (1) a careful encoding of subtyping that leverages efficient quantifier instantiation techniques [6], and (2) the use of optional type equality constraints, which considerably reduce the solution search space. Whenever a valid type assignment for the input program cannot be found, Typpete encodes type error localization as an optimization problem [19] and reports on only minimal sets of fulfilled constraints to help the user pinpoint the cause of the error.

The Lyra Static Analyzer

Evaluation Design Tradeoffs in Numeric Static Analysis for Java

Shiyi Wei\textsuperscript{1,3\dagger}, Piotr Mardzic\textsuperscript{2}, Andrew Rue\textsuperscript{2}, Jeffrey S. Foster\textsuperscript{3}, and Michael Hicks\textsuperscript{3}

\textsuperscript{1} The University of Texas at Dallas, Richardson, USA
\textsuperscript{2} Carnegie Mellon University, Moffett Field, USA
\textsuperscript{3} University of Maryland, College Park, USA

\{swei\textsubscript{\textsuperscript{\textregistered}}@utdallas.edu, piotr\textsubscript{m}@gmail.com, avv\textsubscript{uwf}, jfoster\textsubscript{mu}@cs.umd.edu\}

Abstract. Numeric static analysis for Java has a broad range of potentially useful applications, including array bounds checking and resource usage estimation. However, designing a scalable numeric static analysis for real-world Java programs presents a multitude of design choices, each of which may interact with others. For example, an analysis could handle method calls via either a top-down or bottom-up interprocedural analysis. Moreover, this choice could interact with how we choose to represent aliasing in the heap and/or whether we use a relational numeric domain, e.g., convex polyhedra. In this paper, we present a family of abstract interpretation-based numeric static analyses for Java and systematically evaluate the impact of 162 analysis configurations on the DaCapo benchmark suite. Our experiment considered the precision and performance of the analyses for discharging array bounds checks. We found that top-down analysis is generally a better choice than bottom-up analysis, and that using access paths to describe heap objects is better than using summary objects corresponding to points-to analysis locations. Moreover, these two choices are the most significant, while choices about the numeric domain, representation of abstract objects, and context-sensitivity make much less difference to the precision/performance tradeoff.

1 Introduction

Static analysis of numeric program properties has a broad range of useful applications. Such analyses can potentially detect array bounds errors \[50\], analyze a program’s resource usage \[28,30\], detect side channels \[8,11\], and discover vectors for denial of service attacks \[10,26\].

One of the major approaches to numeric static analysis is abstract interpretation \[18\], in which program statements are evaluated over an abstract domain until a fixed point is reached. Indeed, the first paper on abstract interpretation \[18\] used numeric intervals as one example abstract domain.
The Lyra Static Analyzer

The Lyra Static Analyzer is a tool designed for static analysis of Python programs. It comprises several components, including:

- **FRONT-END**
  - TYPE INference
  - CFG GENERATOR
  - CFG INTERPRETER
- **ABSTRACT DOMAINS**
  - NUMERICAL
    - APRON
    - APRONPY

The tool integrates various components to perform type inference and analysis. The main components involve:

- **TYPE INference**
  - Uses Z3 for constraint solving.

The analysis framework is designed to handle various types of analysis, including numerical and abstract domains.

---

**Abstract.** We present Typpete, a tool that automatically infers Python 3 type annotations. Typpete encodes type constraints as a MaxSMT problem and uses optional constraints and specific quantifier instantiation patterns to make the constraint solving process efficient. Our experimental evaluation shows that Typpete scales to real-world Python programs and outperforms state-of-the-art tools.

1. **Introduction**
   Dynamically-typed languages like Python have become increasingly popular in the past five years. Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing offers early error detection, efficient execution, and machine-checked code documentation, and enables more advanced static analysis and verification approaches.

   For these reasons, Python's PEP484 has recently introduced optional type annotations in the spirit of gradual typing. The annotations can be checked using MyPy. In this paper, we present our tool Typpete, which automatically infers sound (non-gradual) type annotations and can therefore serve as a preprocessor for other analysis or verification tools.

   Typpete performs whole-program type inference, as there are no principal typings in object-oriented languages like Python; the inferred types are correct in the given context but may not be as general as possible. The type inference is constraint-based and relies on the off-the-shelf SMT solver Z3 for finding a valid type assignment for the input program.

   We show that two main ingredients allow Typpete to scale to real programs: (1) a careful encoding of subtyping that leverages efficient quantifier instantiation techniques, and (2) the use of optional type equality constraints, which considerably reduce the solution search space. Whenever a valid type assignment for the input program cannot be found, Typpete encodes type error localization as an optimization problem and reports only minimal subsets of fulfilled constraints to help the user pinpoint the cause of the error.

---

© The Author(s) 2018

https://doi.org/10.1007/978-3-319-96142-2_2

---

Typpete is a tool for automatically inferring Python 3 type annotations. It uses Z3 for solving type constraints and supports numerical abstract domains like APRON and APRONPY.
Numerical Abstract Domains

Implemented Natively

Built on APRON

APRON
https://github.com/antoinemine/apron
ApronPy
Python Bindings for the APRON Numerical Abstract Domain Library

typedef struct ap_abstract1_t {
    ap_abstract0_t* abstract0;
    ap_environment_t* env;
} ap_abstract1_t;

bool ap_abstract1_is_bottom(ap_manager_t* man, ap_abstract1_t* a);
bool ap_abstract1_is_top(ap_manager_t* man, ap_abstract1_t* a);
bool ap_abstract1_is_leq(ap_manager_t* man, ap_abstract1_t* a1, ap_abstract1_t* a2);
bool ap_abstract1_is_eq(ap_manager_t* man, ap_abstract1_t* a1, ap_abstract1_t* a2);

class Abstract1(Structure):
    _fields_ = [
        ('abstract0', POINTER(Abstract0)),
        ('env', POINTER(Environment))
    ]

class PyAbstract1(metaclass=ABCMeta):
    ...

def is_bottom(self):
    return bool(libapron.ap_abstract1_is_bottom(self.manager, self))

def is_top(self):
    return bool(libapron.ap_abstract1_is_top(self.manager, self))

def is_leq(self, other: 'PyAbstract1'):
    return bool(libapron.ap_abstract1_is_leq(self.manager, self, other))

def is_eq(self, other: 'PyAbstract1'):
    return bool(libapron.ap_abstract1_is_eq(self.manager, self, other))

https://github.com/caterinaurban/apronpy
The Lyra Static Analyzer

FRONT-END

https://github.com/caterinaurban/Typpete

TYPE INFERENANCE

Z3
https://github.com/Z3Prover/z3

CFG GENERATOR

CFG INTERPRETER

ABSTRACT DOMAINS

STRING

NUMERICAL

APRON
https://github.com/antoinemine/apron

APRONPY
https://github.com/caterinaurban/apronpy

© The Author(s) 2018
https://doi.org/10.1007/978-3-319-96142-2

Abstract. We present Typpete, an automated type inference tool for Python 3. Typpete encodes type constraints as a MaxSMT problem and uses optional constraints and specific quantifier instantiation patterns to make the constraint solving process efficient. Our experimental evaluation shows that Typpete scales to real-world Python programs and outperforms state-of-the-art tools.

1 Introduction

Dynamically-typed languages like Python have become increasingly popular in the past five years. Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing offers early error detection, efficient execution, and machine-checked code documentation, and enables more advanced static analysis and verification approaches [15]. For these reasons, Python's PEP484 [25] has recently introduced optional type annotations in the spirit of gradual typing [23]. The annotations can be checked using MyPy [10]. In this paper, we present our tool Typpete, which automatically infers sound (non-gradual) type annotations and can therefore serve as a preprocessor for other analysis or verification tools.

Typpete performs whole-program type inference, as there are no principal typings in object-oriented languages like Python [1], example in Sect. 1; the inferred types are correct in the given context but may not be as general as possible. The type inference is constraint-based and relies on the off-the-shelf SMT solver Z3 [7] for finding a valid type assignment for the input program. We show that two main ingredients allow Typpete to scale to real programs: (1) a careful encoding of subtyping that leverages efficient quantifier instantiation techniques [6], and (2) the use of optional type equality constraints, which considerably reduce the solution search space. Whenever a valid type assignment for the input program cannot be found, Typpete encodes type error localization as an optimization problem [19] and reports only minimal sets of fulfilled constraints to help the user pinpoint the cause of the error.

© The Author(s) 2018
https://doi.org/10.1007/978-3-319-96142-2

Abstract. We present Typpete, an automated type inference tool for Python 3. Typpete encodes type constraints as a MaxSMT problem and uses optional constraints and specific quantifier instantiation patterns to make the constraint solving process efficient. Our experimental evaluation shows that Typpete scales to real-world Python programs and outperforms state-of-the-art tools.

1 Introduction

Dynamically-typed languages like Python have become increasingly popular in the past five years. Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing offers early error detection, efficient execution, and machine-checked code documentation, and enables more advanced static analysis and verification approaches [15]. For these reasons, Python's PEP484 [25] has recently introduced optional type annotations in the spirit of gradual typing [23]. The annotations can be checked using MyPy [10]. In this paper, we present our tool Typpete, which automatically infers sound (non-gradual) type annotations and can therefore serve as a preprocessor for other analysis or verification tools.

Typpete performs whole-program type inference, as there are no principal typings in object-oriented languages like Python [1], example in Sect. 1; the inferred types are correct in the given context but may not be as general as possible. The type inference is constraint-based and relies on the off-the-shelf SMT solver Z3 [7] for finding a valid type assignment for the input program. We show that two main ingredients allow Typpete to scale to real programs: (1) a careful encoding of subtyping that leverages efficient quantifier instantiation techniques [6], and (2) the use of optional type equality constraints, which considerably reduce the solution search space. Whenever a valid type assignment for the input program cannot be found, Typpete encodes type error localization as an optimization problem [19] and reports only minimal sets of fulfilled constraints to help the user pinpoint the cause of the error.

© The Author(s) 2018
https://doi.org/10.1007/978-3-319-96142-2
A suite of abstract domains for static analysis of string values

Giulia Costantini†, Pietro Ferrara2,3 and Agostino Cortesi2
1Ca' Foscari University, Venice, Italy
2ETH Zurich, Switzerland
3IBM Thomas J. Watson Research Center, USA

SUMMARY

Strings are widely used in modern programming languages in various scenarios. For instance, strings are used to build up Structured Query Language (SQL) queries that are then executed. Malformed strings may lead to subtle bugs, as well as non-sanitized strings may cause security issues in an application. For these reasons, the application of static analysis to compute safety properties over string values at compile time is particularly appealing. In this article, we propose a generic approach for the static analysis of string values based on abstract interpretation. In particular, we design a suite of abstract semantics for strings, where each abstract domain tracks a different kind of information. We discuss the trade-off between efficiency and accuracy when using such domains to catch the properties of interest. In this way, the analysis can be tuned at different levels of precision and efficiency, and it can address specific properties. Copyright © 2013 John Wiley & Sons Ltd.

Received 23 November 2012; Revised 8 May 2013; Accepted 16 July 2013

KEY WORDS: static analysis; abstract interpretation; abstract domains; strings

Strings are widely used in modern programming languages, such as JavaScript or Python. They are used to build up SQL queries that are then executed. Malformed strings may lead to subtle bugs, as well as non-sanitized strings may cause security issues in an application. For these reasons, the application of static analysis to compute safety properties over string values at compile time is particularly appealing. In this article, we propose a generic approach for the static analysis of string values based on abstract interpretation. In particular, we design a suite of abstract semantics for strings, where each abstract domain tracks a different kind of information. We discuss the trade-off between efficiency and accuracy when using such domains to catch the properties of interest. In this way, the analysis can be tuned at different levels of precision and efficiency, and it can address specific properties.

Static Program Analysis for String Manipulation Languages

Vincenzo Arceri
University of Verona, Verona, Italy
vincenzo.arceri@univr.it

Isabella Mastroeni
University of Verona, Verona, Italy
isabella.mastroeni@univr.it

In recent years, dynamic languages, such as JavaScript or Python, have been increasingly used in a wide range of fields and applications. Their tricky and misunderstood behaviors pose a hard challenge for static analysis of these programming languages. A key aspect of any dynamic language program is the multiple usage of strings, since they can be implicitly converted to another type value, transformed by string-to-code primitives or used to access an object property. Unfortunately, string analyses for dynamic languages still lack precision and do not take into account some important string features. Moreover, string obfuscation is very popular in the context of dynamic language malicious code, for example, to hide code information inside strings and then to dynamically transform strings into executable code. In this scenario, more precise string analyses become a necessity. This paper is placed in the context of static string analysis by abstract interpretation and proposes a new semantics for string analysis, placing a first step for handling dynamic languages string features.

1 Introduction

Dynamic languages, such as JavaScript or Python, have faced an important increment of usage in a very wide range of fields and applications. Common features in dynamic languages are dynamic typing (typing occurs during program execution, at run-time) and implicit type conversion [58], lightening the development phase and allowing not to block the program execution in presence of unexpected or unpre-
Abstract.
We present Typpete, as our tool automatically infers Python 3 type annotations. Typpete encodes type constraints as a MaxSMT problem and uses optional constraints and specific quantifier instantiation patterns to make the constraint solving process efficient. Our experimental evaluation shows that Typpete scales to real world Python programs and outperforms state-of-the-art tools.

Introduction
Dynamically-typed languages like Python have become increasingly popular in the past five years. Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing offers early error detection, efficient execution, and machine-checked code documentation, and enables more advanced static analysis and verification approaches. For these reasons, Python's PEP484 has recently introduced optional type annotations in the spirit of gradual typing. The annotations can be checked using MyPy. In this paper, we present our tool Typpete, which automatically infers sound (non-gradual) type annotations and can therefore serve as a preprocessor for other analysis or verification tools.

Typpete performs whole-program type inference, as there are no principal typings in object-oriented languages like Python; the inferred types are correct in the given context but may not be as general as possible. The type inference is constraint-based and relies on the off-the-shelf SMT solver Z3 for finding a valid type assignment for the input program. We show that two main ingredients allow Typpete to scale to real programs: (1) a careful encoding of subtyping that leverages efficient quantifier instantiation techniques, and (2) the use of optional type equality constraints, which considerably reduce the solution search space. Whenever a valid type assignment for the input program cannot be found, Typpete encodes type error localization as an optimization problem and reports only minimal sets to fulfill constraints to help the user pinpoint the cause of the error.
Container Abstract Domains

Default: Expansion + Summarization

Lists

Dictionaries

Numeric Domains with Summarized Dimensions

Denis Gopan¹, Frank DiMaio², Naurit Dor³, Thomas Reps¹, and Mooya Sagiv²

¹ Comp. Sci. Dept., University of Wisconsin, {gopan, dimaio, reps}@cs.wisc.edu
² School of Comp. Sci., Tel-Aviv University, {murr, mooya}@post.tau.ac.il

Abstract. We introduce a systematic approach to designing summarizing abstract numeric domains from existing numeric domains. Summarizing domains use summary dimensions to represent potentially unbounded collections of numeric objects. Such domains are of benefit to analyses that verify properties of systems with an unbounded number of numeric objects, such as shape analysis, or systems in which the number of numeric objects is bounded, but large.

1 Introduction

Verifying the correctness of complex software systems requires reasoning about numeric quantities. In particular, an analysis technique may have to discover certain relationships among values of numeric objects, such as numeric variables, numeric array elements, or numeric-valued fields of heap-allocated structures [2]. For example, to verify that there are no buffer overruns in a particular C program, an analysis needs to make sure that the value of an index variable does not exceed the length of the buffer at each program point where the buffer is accessed [16].

Numeric analyses have been a research topic for several decades, and a number of numeric domains that allow to approximate numeric state of a system have been...
The Lyra Static Analyzer

**FRONT-END**
- TYPE INFEREN ce
- CFG GENERATOR
- CFG INTERPRETER

**ABSTRACT DOMAINS**
- DATA SCIENCE
- CONTAINER
- STRING
- NUMERICAL
- APRON
- APRONPY

**TYPE INFERENCE**
- https://github.com/caterinaurban/Typpete
- Z3
  - https://github.com/Z3Prover/z3

**CFG GENERATOR**

**CFG INTERPRETER**
- ABSTRACT DOMAIN
  - FORWARD/BACKWARD
  - INTRA/INTER-PROCEDURAL

**ANALYSIS ENGINE**

---

**Abstract.** We present Typpete, as our tool automatically infers Python 3 type annotations. Typpete encodes type constraints as a MaxSMT problem and uses optional constraints and specific quantifier instantiation patterns to make the constraint solving process efficient. Our experimental evaluation shows that Typpete scales to real world Python programs and outperforms state-of-the-art tools.

1 Introduction

Dynamically-typed languages like Python have become increasingly popular in the past five years. Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing offers early error detection, efficient execution, and machine-checked code documentation, and enables more advanced static analysis and verification approaches [15].

For these reasons, Python's PEP484 [25] has recently introduced optional type annotations in the spirit of gradual typing [23]. The annotations can be checked using MyPy [10]. In this paper, we present our tool Typpete, which automatically infers sound (non-gradual) type annotations and can therefore serve as a preprocessor for other analysis or verification tools.

Typpete performs whole-program type inference, as there are no principal typings in object-oriented languages like Python [1], example in Sect. 1; the inferred types are correct in the given context but may not be as general as possible. The type inference is constraint-based and relies on the off-the-shelf SMT solver Z3 [7] for finding a valid type assignment for the input program.

We show that two main ingredients allow Typpete to scale to real programs: (1) a careful encoding of subtyping that leverages efficient quantifier instantiation techniques [6], and (2) the use of optional type equality constraints, which considerably reduce the solution search space. Whenever a valid type assignment for the input program cannot be found, Typpete encodes type error localization as an optimization problem [19] and reports on only minimal sets to fulfill all constraints to help the user pinpoint the cause of the error.

© The Author(s) 2018 H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10982, pp. 12–19, 2018. https://doi.org/10.1007/978-3-319-96142-2_2
Data Usage Abstract Domains

pre-processing → training → data analysis

Data Science

- mislabeled data
- accidentally duplicated data
- wrongly converted data
- accidentally (un)used data
Data Usage Abstract Domains

Growth in a Time of Debt

By Carmen M. Reinhart and Kenneth S. Rogoff

<table>
<thead>
<tr>
<th>Country</th>
<th>Coverage</th>
<th>30 or less</th>
<th>30 to 60</th>
<th>60 to 90</th>
<th>90 or above</th>
<th>30 or less</th>
<th>30 to 60</th>
<th>60 to 90</th>
<th>90 or above</th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
<td>1946-2009</td>
<td>3.4</td>
<td>3.3</td>
<td>-2.0</td>
<td>n.a.</td>
<td>3.4</td>
<td>3.3</td>
<td>-2.0</td>
<td>n.a.</td>
</tr>
<tr>
<td>UK</td>
<td>1946-2009</td>
<td>3.4</td>
<td>2.4</td>
<td>2.5</td>
<td>2.4</td>
<td>3.6</td>
<td>2.9</td>
<td>2.7</td>
<td>6.3</td>
</tr>
<tr>
<td>Sweden</td>
<td>1946-2009</td>
<td>3.6</td>
<td>2.9</td>
<td>2.7</td>
<td>n.a.</td>
<td>3.6</td>
<td>2.9</td>
<td>2.7</td>
<td>6.3</td>
</tr>
<tr>
<td>Spain</td>
<td>1946-2009</td>
<td>1.5</td>
<td>3.4</td>
<td>4.2</td>
<td>n.a.</td>
<td>1.5</td>
<td>3.4</td>
<td>4.2</td>
<td>9.9</td>
</tr>
<tr>
<td>Portugal</td>
<td>1952-2009</td>
<td>4.8</td>
<td>2.5</td>
<td>0.3</td>
<td>n.a.</td>
<td>4.8</td>
<td>2.5</td>
<td>0.3</td>
<td>7.9</td>
</tr>
<tr>
<td>New Zealand</td>
<td>1948-2009</td>
<td>2.5</td>
<td>2.9</td>
<td>3.9</td>
<td>-7.9</td>
<td>2.5</td>
<td>2.9</td>
<td>3.9</td>
<td>-7.9</td>
</tr>
<tr>
<td>Netherlands</td>
<td>1956-2009</td>
<td>4.1</td>
<td>2.7</td>
<td>1.1</td>
<td>n.a.</td>
<td>4.1</td>
<td>2.7</td>
<td>1.1</td>
<td>6.4</td>
</tr>
<tr>
<td>Norway</td>
<td>1947-2009</td>
<td>3.4</td>
<td>5.1</td>
<td>n.a.</td>
<td>n.a.</td>
<td>3.4</td>
<td>5.1</td>
<td>n.a.</td>
<td>5.4</td>
</tr>
<tr>
<td>Japan</td>
<td>1946-2009</td>
<td>7.0</td>
<td>4.0</td>
<td>1.0</td>
<td>0.7</td>
<td>7.0</td>
<td>4.0</td>
<td>1.0</td>
<td>0.7</td>
</tr>
<tr>
<td>Italy</td>
<td>1951-2009</td>
<td>5.4</td>
<td>2.1</td>
<td>1.8</td>
<td>1.0</td>
<td>5.4</td>
<td>2.1</td>
<td>1.8</td>
<td>1.0</td>
</tr>
<tr>
<td>Germany</td>
<td>1946-2009</td>
<td>3.9</td>
<td>0.9</td>
<td>n.a.</td>
<td>n.a.</td>
<td>3.9</td>
<td>0.9</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>France</td>
<td>1949-2009</td>
<td>4.9</td>
<td>2.7</td>
<td>3.0</td>
<td>n.a.</td>
<td>4.9</td>
<td>2.7</td>
<td>3.0</td>
<td>5.2</td>
</tr>
<tr>
<td>Finland</td>
<td>1946-2009</td>
<td>3.8</td>
<td>2.4</td>
<td>5.5</td>
<td>n.a.</td>
<td>3.8</td>
<td>2.4</td>
<td>5.5</td>
<td>7.0</td>
</tr>
<tr>
<td>Denmark</td>
<td>1950-2009</td>
<td>3.5</td>
<td>1.7</td>
<td>2.4</td>
<td>n.a.</td>
<td>3.5</td>
<td>1.7</td>
<td>2.4</td>
<td>5.6</td>
</tr>
<tr>
<td>Canada</td>
<td>1951-2009</td>
<td>1.9</td>
<td>3.6</td>
<td>4.4</td>
<td>n.a.</td>
<td>1.9</td>
<td>3.6</td>
<td>4.4</td>
<td>2.2</td>
</tr>
<tr>
<td>Belgium</td>
<td>1947-2009</td>
<td>n.a.</td>
<td>n.a.</td>
<td>2.6</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>2.6</td>
<td>n.a.</td>
</tr>
<tr>
<td>Austria</td>
<td>1948-2009</td>
<td>5.2</td>
<td>2.3</td>
<td>3.8</td>
<td>n.a.</td>
<td>5.2</td>
<td>2.3</td>
<td>3.8</td>
<td>5.7</td>
</tr>
<tr>
<td>Australia</td>
<td>1951-2009</td>
<td>3.2</td>
<td>4.9</td>
<td>4.9</td>
<td>n.a.</td>
<td>3.2</td>
<td>4.9</td>
<td>4.9</td>
<td>5.9</td>
</tr>
</tbody>
</table>

Total gross external debt includes the external debts of public and external debt. The threshold for total gross external debt in excess of 90 percent of GDP, declines by about two percent; for levels of 200 percent of GDP, with external debt levels in advanced countries now average nearly 2000. We do note, however, that external debt data incorporate over 3,700 annual observations of public debt data even for many rich countries, and to get more than two or three decades of public debt data even for many rich countries, and to get more than two or three decades of public debt data even for many rich countries, and to get more than two or three decades of public and external debt. The authors would like to thank Olivier Jeanne and Vincent R. Reinhart: Department of Economics, 4115 Tydings Hall, College Park, MD 20742–3001. The Lyra Static Analyzer: Beta release 2009a, b. The authors thank the organizations that have made data available for the project including the World Bank, the International Monetary Fund, the Organization for Economic Co-operation and Development, the International Financial Corporation, the Bank for International Settlements, and the European Commission. The authors would like to thank Olivier Jeanne and Vincent R. Reinhart: Department of Economics, 4115 Tydings Hall, College Park, MD 20742–3001. The Lyra Static Analyzer: Beta release 2009a, b. The authors thank the organizations that have made data available for the project including the World Bank, the International Monetary Fund, the Organization for Economic Co-operation and Development, the International Financial Corporation, the Bank for International Settlements, and the European Commission.

The focus of this paper is on the longer term consequences of debt. These are the subject of an earlier paper (Reinhart and Rogoff 2009b). This paper focuses on the shorter run consequences of debt. The Lyra Static Analyzer: Beta release 2009a, b. The authors thank the organizations that have made data available for the project including the World Bank, the International Monetary Fund, the Organization for Economic Co-operation and Development, the International Financial Corporation, the Bank for International Settlements, and the European Commission. The authors would like to thank Olivier Jeanne and Vincent R. Reinhart: Department of Economics, 4115 Tydings Hall, College Park, MD 20742–3001. The Lyra Static Analyzer: Beta release 2009a, b. The authors thank the organizations that have made data available for the project including the World Bank, the International Monetary Fund, the Organization for Economic Co-operation and Development, the International Financial Corporation, the Bank for International Settlements, and the European Commission.
Data Usage Abstract Domains

Growth in a Time of Debt

By Carmen M. Reinhart and Kenneth S. Rogoff*

<table>
<thead>
<tr>
<th>Country</th>
<th>Coverage</th>
<th>30 or less</th>
<th>30 to 60</th>
<th>60 to 90</th>
<th>90 or above</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
<td>1946-2009</td>
<td>3.4</td>
<td>3.3</td>
<td>-2.0</td>
<td>n.a.</td>
<td>2.0</td>
</tr>
<tr>
<td>UK</td>
<td>1946-2009</td>
<td>2.4</td>
<td>2.5</td>
<td>2.5</td>
<td>n.a.</td>
<td>2.5</td>
</tr>
<tr>
<td>Sweden</td>
<td>1946-2009</td>
<td>3.6</td>
<td>2.9</td>
<td>2.7</td>
<td>n.a.</td>
<td>2.8</td>
</tr>
<tr>
<td>Spain</td>
<td>1946-2009</td>
<td>1.5</td>
<td>3.4</td>
<td>4.2</td>
<td>n.a.</td>
<td>2.9</td>
</tr>
<tr>
<td>Portugal</td>
<td>1952-2009</td>
<td>4.8</td>
<td>2.5</td>
<td>0.3</td>
<td>n.a.</td>
<td>1.6</td>
</tr>
<tr>
<td>New Zealand</td>
<td>1948-2009</td>
<td>2.5</td>
<td>2.9</td>
<td>3.9</td>
<td>n.a.</td>
<td>2.9</td>
</tr>
<tr>
<td>Netherlands</td>
<td>1956-2009</td>
<td>4.1</td>
<td>2.7</td>
<td>1.1</td>
<td>n.a.</td>
<td>2.4</td>
</tr>
<tr>
<td>Norway</td>
<td>1947-2009</td>
<td>3.4</td>
<td>1.1</td>
<td>0.8</td>
<td>n.a.</td>
<td>1.3</td>
</tr>
<tr>
<td>Japan</td>
<td>1946-2009</td>
<td>7.0</td>
<td>4.0</td>
<td>1.0</td>
<td>n.a.</td>
<td>3.7</td>
</tr>
<tr>
<td>Italy</td>
<td>1951-2009</td>
<td>5.4</td>
<td>2.1</td>
<td>1.8</td>
<td>n.a.</td>
<td>3.1</td>
</tr>
<tr>
<td>Ireland</td>
<td>1948-2009</td>
<td>4.4</td>
<td>4.5</td>
<td>4.0</td>
<td>n.a.</td>
<td>4.4</td>
</tr>
<tr>
<td>Greece</td>
<td>1970-2009</td>
<td>4.0</td>
<td>0.3</td>
<td>3.0</td>
<td>n.a.</td>
<td>2.3</td>
</tr>
<tr>
<td>Germany</td>
<td>1946-2009</td>
<td>3.9</td>
<td>0.9</td>
<td>3.9</td>
<td>n.a.</td>
<td>2.6</td>
</tr>
<tr>
<td>France</td>
<td>1949-2009</td>
<td>4.9</td>
<td>2.7</td>
<td>3.0</td>
<td>n.a.</td>
<td>3.3</td>
</tr>
<tr>
<td>Finland</td>
<td>1944-2009</td>
<td>3.8</td>
<td>2.4</td>
<td>5.5</td>
<td>n.a.</td>
<td>3.6</td>
</tr>
<tr>
<td>Denmark</td>
<td>1950-2009</td>
<td>3.5</td>
<td>1.7</td>
<td>2.4</td>
<td>n.a.</td>
<td>2.4</td>
</tr>
<tr>
<td>Canada</td>
<td>1951-2009</td>
<td>1.9</td>
<td>1.0</td>
<td>4.1</td>
<td>n.a.</td>
<td>2.3</td>
</tr>
<tr>
<td>Belgium</td>
<td>1947-2009</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Austria</td>
<td>1951-2009</td>
<td>3.2</td>
<td>4.9</td>
<td>4.0</td>
<td>n.a.</td>
<td>3.8</td>
</tr>
<tr>
<td>Australia</td>
<td>1951-2009</td>
<td>3.2</td>
<td>4.9</td>
<td>4.0</td>
<td>n.a.</td>
<td>3.8</td>
</tr>
</tbody>
</table>

FAQ: Reinhart, Rogoff, and the Excel Error That Changed History

By Peter Coy  April 18, 2013

The Excel Depression

By Paul Krugman  Published: April 18, 2013  470 Comments

In this age of information, math errors can lead to disaster. NASA’s Mars Orbiter crashed because engineers forgot to convert to metric measurements; JPMorgan Chase’s “London Whale” venture went bad in part because modelers divided by a sum instead of an average. So, did an Excel coding error destroy the economies of the Western world?

The story so far: At the beginning of 2010, two Harvard economists, Carmen Reinhart and Kenneth Rogoff, circulated a paper, “Growth in a Time of Debt,” that purported to identify a critical “threshold,” a tipping point, for government indebtedness. Once debt exceeds 90 percent of gross domestic product, they claimed, economic growth drops off sharply.

Ms. Reinhart and Mr. Rogoff had credibility thanks to a widely admired earlier book on the history of financial
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 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
Data Usage Abstract Domains

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 critical role in decision making in fields ranging from economy to biology and medicine. As a result, errors in data science can have severe consequences, especially when they lead to decisions that look plausible, but are incorrect. A common cause of such erroneous but plausible results is accidental duplication of data, which was widely cited in political debates and academic journals.

In this paper, we propose an abstract interpretation framework that automatically detects unused input data. We derive a precise definition of a directed acyclic graph (DAG) that precisely captures data usage by abstraction of the (compositional) trace semantics and express it in a constructive fixed point form.

This clear design principle provides a framework that subsumes existing analyses: we show that secure information flow analysis, pre-processing, and post-processing as well as a recent work that uses abstract computation graphs (ACGs) to find strongly-live variables analysis can be used for data usage, with precision. Additionally, we derive a static analysis for unused input data, which is similar to dependency analysis and context of backward program slicing. Finally, we demon state express such analyses as abstract interpretation by with an existing abstraction of compound data structures and lists to detect used 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...
Data Shape Abstract Domains
Implicit Assumptions on the Input Data

The Lyra Static Analyzer

Data Shape Abstract Domains

Data science software plays an increasingly important role in many fields, ranging from economy and finance to biology and medicine. As we rely more and more on data science for making critical decisions, the need for robust tools to ensure the accuracy and reliability of the results becomes paramount.

Implicit Assumptions on the Input Data

When processing data, it is crucial to ensure that the input data is consistent and accurate. Implicit assumptions often underlie the design of data science applications, and these assumptions can sometimes lead to unexpected results. For instance, consider the following Python code snippet:

```python
import sys

grade2gpa = {'A': 4.0, 'B': 3.0, 'C': 2.0, 'D': 1.0, 'F': 0.0}
with open(sys.argv[1]) as file:
    for line in file:
        data = line.strip().split(' ')
        grades = int(data[1])
        gpa = 0.0
        for i in range(2, grades + 2):
            gpa += grade2gpa[data[i]]
        result = gpa / grades
        print('{}: {}'.format(data[0], result))

```

In this example, the code assumes that the input data is in a specific format and that the grades are integer values. If the input data does not conform to these assumptions, the program may produce incorrect results. For instance, if the input data includes non-integer grades or if the format is incorrect, the program may fail or produce unexpected output.

To mitigate these risks, static analysis tools like the Lyra Static Analyzer can be used. These tools automatically infer sound (non-gradual) type annotations and can therefore help pinpoint the cause of the error. For example, the Lyra Static Analyzer can check that the input program cannot be found, thereby considerably reducing the solution search space.

This clear design principle provides a framework that subsumes existing abstractions of compound data structures such as arrays. Based on this semantics, we systematically derive static analyses that automatically detect unused input data by fixpoint approximation. In this paper, we propose an abstract interpretation framework to perform whole-program type inference, as there are no principled approaches to scale to real programs. We demonstrate the value of precision. Additionally, we derive a static analysis to detect single live variables analysis can be used for data usage, with varying degrees of precision.

Based on this treatment, we systematically derive static analyses that automatically detect unused input data by fixpoint approximation. In this paper, we propose an abstract interpretation framework to perform whole-program type inference, as there are no principled approaches to scale to real programs. We demonstrate the value of precision. Additionally, we derive a static analysis to detect single live variables analysis can be used for data usage, with varying degrees of precision.

In conclusion, implicit assumptions on the input data can lead to unexpected results if not properly addressed. Static analysis tools like the Lyra Static Analyzer can help ensure the accuracy and reliability of data science applications by automatically detecting issues and providing insights into the source of the problems.
Data Shape Abstract Domains

Implicit Assumptions on the Input Data
The Lyra Static Analyzer

**ANALYSIS ENGINE**

**FRONT-END**

- **TYPE INFERENCE**
  - Z3
    - https://github.com/Z3Prover/z3
  - Typpete
    - https://github.com/caterinaurban/Typpete

- **CFG GENERATOR**

- **CFG INTERPRETER**
  - ABSTRACT DOMAIN
    - FORWARD/BACKWARD
    - INTRA/INTER-PROCEDURAL

**ABSTRACT DOMAINS**

- **DATA SCIENCE**

- **CONTAINER**

- **STRING**

- **NUMERICAL**
  - APRON
    - https://github.com/antoinemine/apron
  - APRONPY
    - https://github.com/caterinaurban/apronpy

- **APRON**
  - https://github.com/caterinaurban/apronpy

- **https://doi.org/10.1007/978-3-319-96142-2
  © The Author(s) 2018

---

**Introduction**

Programming errors can cause frustration, especially when they lead to a programming failure after hours of computation. However, programming errors that do not cause failures can have more serious consequences as code that produces erroneous but plausible results gives no indication that something went wrong. Professionals outside the software development community often do not understand the potential for significant errors or any repercussions that could result from them.

Program failure after hours of computation is usually not a problem on its own: it is a symptom of someone being careless, and this error can be fixed by making the code more robust. However, programming errors that do not cause failures can have more serious consequences. For example, a mathematician working on a proof that a program runs in polynomial time might have spent a considerable amount of time discovering and implementing a new, efficient algorithm. Then, due to a programming error, the result might be not polynomial at all. This would have two implications:

1. Wasting a lot of time on research.
2. Possibly proving a theorem that is false.

In critical decision making in fields ranging from economy and finance to biology and medicine, errors in data science applications erroneously ignore some of their input data, for instance due to bugs in the code that reads, filters, or clusters it. This can have severe consequences, especially when they lead to results that are used as input to critical decision making.

In the past few years, data science has grown considerably in importance and has become an increasingly important role in our society. While historically economists>Your name</name> and economists Reinhart and Rogoff on how they introduced an erroneous but plausible result gives no indication that something went wrong. For these reasons, Python's PEP484 has recently introduced optional type equality constraints, which con-
The Lyra Static Analyzer for Data Science Software

The Author(s) 2018

Mostafa Hassan

Department of Computer Science, ETH Zurich, Zurich, Switzerland

Caterina Urban

Department of Computer Science, ETH Zurich, Zurich, Switzerland

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 683–710, 2018. © The Author(s) 2018

Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing o...