Neural Software Analysis: Learning Developer Tools from Code

Michael Pradel
Software Lab – University of Stuttgart
Joint work with Jibesh Patra, Georgios Gousios, Jason Liu, and Satish Chandra
Developers Need Tools

Key feature of humans: Ability to develop tools

Software development tools, e.g., compilers, bug detection, code completion, documenting software
Creating Developer Tools

Traditional program analysis

- Manually crafted
- Years of work
- Precise, logical reasoning
- Heuristics to handle undecidability
- Challenged by large code bases
Creating Developer Tools

**Traditional program analysis**
- Manually crafted
- Years of work
- Precise, logical reasoning
- Heuristics to handle undecidability
- Challenged by large code bases

**Neural software analysis**
- Automatically learned within hours
- Data-driven prediction
- Learn instead of hard-code heuristics
- Use big code to our benefit
Learning Developer Tools

Insight: Lots of **data** about software development to learn from

Source code
Execution traces
Documentation
Bug reports
etc.

[Diagram]

Machine Learning

Predictive tool
Learning Developer Tools

Insight: Lots of **data about software development** to learn from

- Source code
- Execution traces
- Documentation
- Bug reports
- etc.

Predictive tool

New code, execution, etc.

Information useful for developers

Machine Learning
This Talk

Neural software analysis

1) Nalin: Name-value inconsistencies

2) TypeWriter: Type prediction

More details:
- Neural Software Analysis, CACM 2022
- Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks, ICSE 2022
- TypeWriter: Neural Type Prediction with Search-based Validation, FSE 2020
train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]
Motivation

Incorrect value: 135.0, should be 135

```
train_size = 0.9 * iris.data.shape[0]

test_size = iris.data.shape[0] - train_size

train_data = data[0:train_size]
```
Motivation

Incorrect value:
135.0, should be 135

```python
train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]
```

```python
def file = os.path.exists('reference.csv')
if file == False:
    print('Warning: ...')
```
Motivation

Incorrect value:
135.0, should be 135

\[
\begin{align*}
\text{train\_size} & = 0.9 \times \text{iris\_data\_shape}[0] \\
\text{test\_size} & = \text{iris\_data\_shape}[0] - \text{train\_size} \\
\text{train\_data} & = \text{data}[0:\text{train\_size}] \\
\end{align*}
\]

Misleading name:
file vs. boolean
Motivation

Incorrect value:
135.0, should be 135

Commonality:
Name and value are inconsistent

train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]

file = os.path.exists('reference.csv')
if file == False:
    print('Warning: ...')

Misleading name:
file vs. boolean
Goal

Finding name-value inconsistencies
Goal

Challenge 1: Understand the meaning of names

Finding name-value inconsistencies
Goal

Challenge 1: Understand the meaning of names

Challenge 2: Understand the meaning of values

Finding name-value inconsistencies
Goal

Challenge 1: Understand the meaning of names

Finding name-value inconsistencies

Challenge 2: Understand the meaning of values

Challenge 3: Precisely pinpoint unusual pairs
Overview of Nalin

Training
- Generation of negative examples
- Dynamic analysis of assignments
- Train neural model
- Query neural model
- Name-value inconsistencies

Executable programs

Prediction
Analyzing Assignments

Option 1: Static analysis

■ 90% of assignments:
  Complex expression as right-hand side

\[
\text{train\_size} = 0.9 \times \text{iris.data.shape}[0]
\]

\[
\text{file} = \text{os.path.exists(’reference.csv’)}
\]
Analyzing Assignments

Option 1: Static analysis

- 90% of assignments:
  Complex expression as right-hand side

```python
train_size = 0.9 * iris.data.shape[0]
file = os.path.exists('reference.csv')
```

Difficult to obtain exact value
Analyzing Assignments

Option 2: **Dynamic analysis**

- Source-to-source instrumentation
- Extract for each assignment
  - Name of left-hand side
  - String representation of value
  - Type of value
  - Length of value
  - Shape of value
## Analyzing Assignments

### Example:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
<th>Length</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>23</td>
<td>int</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>probability</td>
<td>0.83</td>
<td>float</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>Xs_train</td>
<td>[[0.5 2.3] \ n [ .. ndarray 600 (600,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>2.5</td>
<td>float</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>file_name</td>
<td>'example.txt'</td>
<td>str</td>
<td>11</td>
<td>null</td>
</tr>
</tbody>
</table>
Generation of Negative Examples

**Naive approach**: Randomly combine names and values

Positive examples
- num = 23
- age = 3

Negative examples
- num = 3
- age = 23
Generation of Negative Examples

Naive approach: Randomly combine names and values

Positive examples
num=23
age=3

Negative examples
num=3
age=23

Legitimate “negative” examples:
Noisy training data → False positives
Generation of Negative Examples

Type-guided generation:

1) Given a name, select a type that is
   ■ unusual for this name
   ■ common across all observed values

2) Pick a random value of this type
Generation of Negative Examples

Type-guided generation:

1) Given a name, select a type that is
   - unusual for this name
   - common across all observed values

2) Pick a random value of this type

Positive example
years=[2011, 2012]
Generation of Negative Examples

Type-guided generation:

1) Given a name, select a type that is
   - unusual for this name
   - common across all observed values

2) Pick a random value of this type

Positive example

years=[2011, 2012]
Generation of Negative Examples

Type-guided generation:

1) Given a name, select a type that is
   - unusual for this name
   - common across all observed values

2) Pick a random value of this type

Positive example

years=[2011, 2012]
Generation of Negative Examples

Type-guided generation:

1) Given a name, select a type that is
   - unusual for this name
   - common across all observed values

2) Pick a random value of this type

Positive example
years=[2011, 2012]

Negative example
years=False
Neural Classification Model

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
<th>Length of value</th>
<th>Shape of value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Feed-forward layers</td>
<td>p(inconsistent)</td>
<td></td>
</tr>
</tbody>
</table>

*Embed with FastText, GRU, CNN*

*One-hot, One-hot, One-hot*  

Two linear layers, 50% dropout, Adam optimizer, batch size=128
Evaluation: Setup

Jupyter notebooks: Closed, executable programs

- Initially, 1M programs
- 106k analyzed programs (7.2MLoC)
- 947k name-value pairs
Effectiveness of Classifier

Threshold for reporting warnings

- Precision
- Recall
- F1
Effectiveness of Classifier

Best F1 score: 89%
Study with Developers

Is a name-value pair easy to understand?

40 name-value pairs, 11 participants, 5-point Likert scale, 56% inter-rate agreement
**Study with Developers**

Is a name-value pair easy to understand?

<table>
<thead>
<tr>
<th>Developer assessment</th>
<th>Nalin’s prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Consistent</td>
</tr>
<tr>
<td>Easy to understand</td>
<td>15</td>
</tr>
<tr>
<td>Hard to understand</td>
<td>5</td>
</tr>
</tbody>
</table>

40 name-value pairs, 11 participants, 5-point Likert scale, 56% inter-rate agreement
**Study with Developers**

Is a name-value pair *easy to understand*?

<table>
<thead>
<tr>
<th>Developer assessment</th>
<th>Nalin’s prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Consistent</td>
</tr>
<tr>
<td>Easy to understand</td>
<td>15</td>
</tr>
<tr>
<td>Hard to understand</td>
<td>5</td>
</tr>
</tbody>
</table>

**precision=80%, recall=76%**

40 name-value pairs, 11 participants, 5-point Likert scale, 56% inter-rate agreement
# Kinds of Inconsistencies

<table>
<thead>
<tr>
<th>Count</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>inspected warnings</td>
</tr>
<tr>
<td>21</td>
<td>misleading names</td>
</tr>
<tr>
<td>2</td>
<td>incorrect values</td>
</tr>
<tr>
<td>7</td>
<td>false positives</td>
</tr>
</tbody>
</table>

- **30 inspected warnings**
  - **21 misleading names**
  - **2 incorrect values**
  - **7 false positives**
Kinds of Inconsistencies

30 inspected warnings

21 misleading names

2 incorrect values

7 false positives

Unusual combination

name = 'Philip K. Dick'

...
Kinds of Inconsistencies

30 inspected warnings

21 misleading names
2 incorrect values
7 false positives

\[
prob = \text{get\_betraying\_probability(information)}
\]

\[
\text{if } prob > 1/2:
\]

\[
\text{return } D
\]

Value: "Corporate"
Kinds of Inconsistencies

30 inspected warnings

21 misleading names
2 incorrect values
7 false positives

```
dwarF = '/Users/iayork/Downloads/dwar\_2013\_2015.txt'
dwar = pd.read_csv(dwarF, sep=' ', header=None)
```

Model doesn’t understand the abbreviation ("F" means "file")
### Comparison with Prior Tools

<table>
<thead>
<tr>
<th>Approach</th>
<th>Total Warnings</th>
<th>Shared with Nalin</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyre</td>
<td>54</td>
<td>1/30</td>
</tr>
<tr>
<td>flake8</td>
<td>1,247</td>
<td>0/30</td>
</tr>
<tr>
<td>DeepBugs</td>
<td>151</td>
<td>0/30</td>
</tr>
</tbody>
</table>

Each tool applied to 30 files with manually inspected Nalin warnings.
### Comparison with Prior Tools

<table>
<thead>
<tr>
<th>Approach</th>
<th>Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>pyre</td>
<td>54</td>
</tr>
<tr>
<td>flake8</td>
<td>1,247</td>
</tr>
<tr>
<td>DeepBugs</td>
<td>151</td>
</tr>
</tbody>
</table>

Each tool applied to 30 files with manually inspected Nalin warnings.

Traditional static checkers
Comparison with Prior Tools

<table>
<thead>
<tr>
<th>Approach</th>
<th>Warnings</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Shared with Nalin</td>
</tr>
<tr>
<td>pyre</td>
<td>54</td>
<td>1/30</td>
</tr>
<tr>
<td>flake8</td>
<td>1,247</td>
<td>0/30</td>
</tr>
<tr>
<td>DeepBugs</td>
<td>151</td>
<td>0/30</td>
</tr>
</tbody>
</table>

Learning-based detector of name-related bugs [OOPSLA’18]

Each tool applied to 30 files with manually inspected Nalin warnings
## Comparison with Prior Tools

<table>
<thead>
<tr>
<th>Approach</th>
<th>Warnings</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Shared with Nalin</td>
</tr>
<tr>
<td>pyre</td>
<td>54</td>
<td>1/30</td>
</tr>
<tr>
<td>flake8</td>
<td>1,247</td>
<td>0/30</td>
</tr>
<tr>
<td>DeepBugs</td>
<td>151</td>
<td>0/30</td>
</tr>
</tbody>
</table>

Each tool applied to 30 files with manually inspected Nalin warnings

Variable `name` holds string and then float
Summary: Nalin

- Automatic detection of name-value inconsistencies
- Learning-based bug detection on runtime behavior
- Type-guided generation of negative examples
This Talk

Neural software analysis

1) Nalin: Name-value inconsistencies

2) TypeWriter: Type prediction

More details:
- Neural Software Analysis, CACM 2022
- Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks, ICSE 2022
- TypeWriter: Neural Type Prediction with Search-based Validation, FSE 2020
Types in Dynamic PLs

■ Dynamically typed languages: Extremely popular

■ Lack of type annotations:
  □ Type errors
  □ Hard-to-understand APIs
  □ Poor IDE support

■ Gradual types to the rescue
Types in Dynamic PLs

- Dynamically typed languages: Extremely popular
- Lack of type annotations:
  - Type errors
  - Hard-to-understand APIs
  - Poor IDE support
- Gradual types to the rescue

But: Annotating types is painful
How to Add Type Annotations?

- **Option 1: Static type inference**
  - Guarantees type correctness, but very limited

- **Option 2: Dynamic type inference**
  - Depends on inputs and misses types

- **Option 3: Probabilistic type prediction**
  - Models learned from existing type annotations
Probabilistic Type Prediction

Neural model to predict types

Prior models:
- Deep Learning Type Inference, FSE’18
- NL2Type: Inferring JavaScript Function Types from Natural Language Information, ICSE’19
Challenges

- **Imprecision**
  - Some predictions are wrong
  - Developers must decide which suggestions to follow

- **Combinatorial explosion**
  - For each missing type: One or more suggestions
  - Exploring all combinations: Practically impossible
Example

def find_match(color):
    """
    Args:
        color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ['red', 'blue', 'green']
Example

def find_match(color):
    """
    Args:
    color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
Example

def find_match(color):
    
    """
    Args:
    color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
def find_match(color):
    
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ['red', 'blue', 'green']

Example

Correct predictions

Predictions:
1) int
2) str
3) bool

Predictions:
1) str
2) Optional[str]
3) None

Predictions:
1) int
2) str
3) bool

Predictions:
1) List[str]
2) List[Optional[str]]
3) str
Overview of TypeWriter

Program with type annotations

Search for consistent types

Feedback-directed search

Static type checker

Probabilistic type prediction

Neural type prediction

Lightweight static analysis

Program

NL info

PL info

Type predictions

Neural type prediction

Lightweight static analysis

Overview of TypeWriter
Extracting NL and PL Info

- **NL information**
  - Names of functions and arguments
  - Function-level comments

- **PL information**
  - Occurrences of the to-be-typed code element
  - Types made available via imports
def find_match(color):
    """
    Args:
        color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
def find_match(color):
    """
    Args:
    color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ['red', 'blue', 'green']
def find_match(color):
    """
    Args:
    color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ['red', 'blue', 'green']
def find_match(color):
    """
    Args:
        color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
def find_match(color):
    '''
    Args:
    color (str): color to match on and return
    '''
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
def find_match(color):
    """
    Args:
    color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
from ab import de
import x.y.z

def find_match(color):
    """
    Args:
    color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
Neural Type Prediction Model

Code tokens → Token embedding → RNN

Identifiers → Word embedding → RNN

Comments → RNN

Available types → one-hot encoded type mask → Hidden layer + Softmax → Type vector
Searching for Consistent Types

- **Top-k predictions for each missing type**
  - Filter predictions using gradual type checker
  - E.g., pyre and mypy for Python, flow for JavaScript

- **Combinatorial search problem**
  - For type slots $S$ and $k$ predictions per slot: $(k + 1)^{|S|}$ possible type assignments
Searching for Consistent Types

- **Top-k predictions for each missing type**
  - Filter predictions using gradual type checker
  - E.g., pyre and mypy for Python, flow for JavaScript

- **Combinatorial search problem**
  - For type slots $S$ and $k$ predictions per slot:
    $$(k + 1)^{|S|}$$ possible type assignments

Too large to explore exhaustively!
Exploring the Search Space

Tree of variants of program $P$

$P_{origTypes}$

$P_{addedTypes1}$  $P_{addedTypes2}$

$P_{addedTypes3}$  $P_{addedTypes4}$

... add, remove, or replace types
Exploring the Search Space

Tree of variants of program $P$

$P_{\text{origTypes}}$

$P_{\text{addedTypes}1}$  $P_{\text{addedTypes}2}$

$P_{\text{addedTypes}3}$  $P_{\text{addedTypes}4}$

Which variants to explore first?

... add, remove, or replace types
Feedback Function

- **Goal:** Minimize missing types without introducing type errors

- **Feedback score** (lower is better):

\[ v \cdot n_{missing} + w \cdot n_{errors} \]
Feedback Function

- **Goal:** Minimize missing types without introducing type errors

- **Feedback score** *(lower is better):*
  
  \[ v \cdot n_{\text{missing}} + w \cdot n_{\text{errors}} \]

  **Default:** \( v = 1, w = 2, \)

  i.e., higher weight for errors
Exploring the Search Space

- **Optimistic**: Add top-most predicted type everywhere and then update types.

- **Greedy or non-greedy**

  - If score decreases, keep the type.
  - Backtrack to avoid local minima.
Example

```python
def find_match(color):
    """
    Args:
        color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
```
Example

def find_match(color):
    
    """
    Args:
        color (str): color to match on and return
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ['red', 'blue', 'green']
```python
def find_match(color):
    
    """
    Args:
    color (str): color to match on and return  
    """
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
```

Predictions:
1) `int`
2) `str`
3) `bool`

Predictions:
1) `str`
2) Optional[`str`]
3) `None`

Predictions:
1) `List[`str`]`
2) `List[`Any`]`
3) `str`
Example

```python
def find_match(color):
    ""
    Args:
        color (str): color to match on and return
    ""
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ['red', 'blue', 'green']
```

Predictions:
1) `List[str]`
2) `List[Any]`
3) `str`

Predictions:
1) `str`
2) `Optional[str]`
3) `None`

Predictions:
1) `int`
2) `str`
3) `bool`
```python
def find_match(color):
    
    Args:
    color (str): color to match on and return
    
    candidates = get_colors()
    for candidate in candidates:
        if color == candidate:
            return color
    return None

def get_colors():
    return ["red", "blue", "green"]
```

Predictions:
1) int
2) str
3) bool

Predictions:
1) str
2) Optional[str]
3) None

Predictions:
1) List[str]
2) List[Any]
3) str
Evaluation: Setup

- Code corpora
  - Facebook’s Python code
  - 5.8 millions lines of open-source code

- Types
  - Millions of argument and return types
  - 6-12% already annotated
  - Trivial types (e.g., type of self) ignored
## Effectiveness of Neural Model

<table>
<thead>
<tr>
<th>Approach</th>
<th>Top-1</th>
<th>Prec</th>
<th>Rec</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeWriter</td>
<td>65%</td>
<td>59%</td>
<td>62%</td>
<td></td>
</tr>
<tr>
<td>NL2Type</td>
<td>59%</td>
<td>55%</td>
<td>57%</td>
<td></td>
</tr>
<tr>
<td>Frequencies</td>
<td>12%</td>
<td>20%</td>
<td>15%</td>
<td></td>
</tr>
</tbody>
</table>

Open-source corpus; Combined prediction (arg. and return types)
## Effectiveness of Neural Model

<table>
<thead>
<tr>
<th>Approach</th>
<th>Top-1 Prec</th>
<th>Top-1 Rec</th>
<th>Top-1 F1</th>
<th>Top-3 Prec</th>
<th>Top-3 Rec</th>
<th>Top-3 F1</th>
<th>Top-5 Prec</th>
<th>Top-5 Rec</th>
<th>Top-5 F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeWriter</td>
<td>65%</td>
<td>59%</td>
<td>62%</td>
<td>80%</td>
<td>71%</td>
<td>75%</td>
<td>85%</td>
<td>75%</td>
<td>80%</td>
</tr>
<tr>
<td>Frequency</td>
<td>12%</td>
<td>20%</td>
<td>15%</td>
<td>19%</td>
<td>35%</td>
<td>25%</td>
<td>22%</td>
<td>39%</td>
<td>28%</td>
</tr>
</tbody>
</table>

Open-source corpus; Combined prediction (arg. and return types)
## Effectiveness of Neural Model

<table>
<thead>
<tr>
<th>Approach</th>
<th>Top-1</th>
<th></th>
<th>Top-3</th>
<th></th>
<th>Top-5</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
<td>F1</td>
<td>Prec</td>
<td>Rec</td>
<td>F1</td>
</tr>
<tr>
<td>TypeWriter</td>
<td>65%</td>
<td>59%</td>
<td>62%</td>
<td>80%</td>
<td>71%</td>
<td>75%</td>
</tr>
<tr>
<td>NL2Type</td>
<td>59%</td>
<td>55%</td>
<td>57%</td>
<td>73%</td>
<td>67%</td>
<td>70%</td>
</tr>
<tr>
<td>Frequencies</td>
<td>12%</td>
<td>20%</td>
<td>15%</td>
<td>19%</td>
<td>35%</td>
<td>25%</td>
</tr>
</tbody>
</table>

Open-source corpus; Combined prediction (arg. and return types)
## Effectiveness of Search

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Top-$k$</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Type-correct</td>
</tr>
<tr>
<td>Greedy search</td>
<td>1</td>
<td>215 (70%)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>230 (75%)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>231 (75%)</td>
</tr>
<tr>
<td>Non-greedy search</td>
<td>1</td>
<td>216 (71%)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>178 (58%)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>164 (54%)</td>
</tr>
<tr>
<td>Pyre Infer</td>
<td>—</td>
<td>106 (35%)</td>
</tr>
</tbody>
</table>

Ground truth: 306 annotations in 47 fully annotated files
Exploring up to $7 \cdot |S|$ states
## Effectiveness of Search

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Top-$k$</th>
<th>Annotations</th>
<th>Type-correct</th>
<th>Ground truth match</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy search</td>
<td>1</td>
<td>215 (70%)</td>
<td>194 (63%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>230 (75%)</td>
<td>196 (64%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>231 (75%)</td>
<td>198 (65%)</td>
<td></td>
</tr>
<tr>
<td>Non-greedy search</td>
<td>1</td>
<td>216 (71%)</td>
<td>195 (64%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>178 (58%)</td>
<td>148 (48%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>164 (54%)</td>
<td>141 (46%)</td>
<td></td>
</tr>
</tbody>
</table>

Ground truth: 306 annotations in 47 fully annotated files
Exploring up to $7 \cdot |S|$ states
## Effectiveness of Search

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Top-$k$</th>
<th>Annotations</th>
<th>Type-correct</th>
<th>Ground truth match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy search</td>
<td>1</td>
<td>215 (70%)</td>
<td>194 (63%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>230 (75%)</td>
<td>196 (64%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>231 (75%)</td>
<td>198 (65%)</td>
<td></td>
</tr>
<tr>
<td>Non-greedy search</td>
<td>1</td>
<td>216 (71%)</td>
<td>195 (64%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>178 (58%)</td>
<td>148 (48%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>164 (54%)</td>
<td>141 (46%)</td>
<td></td>
</tr>
<tr>
<td>Pyre Infer</td>
<td>—</td>
<td>106 (35%)</td>
<td>78 (25%)</td>
<td></td>
</tr>
</tbody>
</table>

Ground truth: 306 annotations in 47 fully annotated files
Exploring up to $7 \cdot |S|$ states
Limitations

- Type-correctness vs. soundness
- Limited type vocabulary
- Gradual type checking (and hence TypeWriter) is relatively slow
Summary: TypeWriter

Neural type prediction with search-based validation

- Probabilistic type prediction based on NL and PL information
- Ensure type correctness of added types via gradual type checker
- TypeWriter tool in use at Facebook
Why Does It Work?

Developers use meaningful names

Source code is repetitive

Many programs available as training data

Probabilistic models + NL = ♥
Neural Software Analysis

When to (not) use it?

- Low fuzziness of available information
- High fuzziness of available information
- Available
- Not available
- Many amount of examples to learn from
- Few amount of examples to learn from
- Well-defined correctness criterion
- Not available

Neural software analysis
Other Interests

Neural software analysis:
- Human attention vs. neural models
- Semantic bug seeding
- Token-level code embeddings
- Ultra-large scale bug localization

Other topics:
- WebAssembly: Security and analysis
- Node.js security
- JavaScript test generation
- Program repair
- Quantum computing platforms
Conclusions

- **Neural software analysis:**  
  More and more code = the problem  
  part of the solution

- **Open challenges**
  - Learning from runtime behavior
  - Clean datasets
  - Combining neural with traditional analysis

Papers, code, etc.: http://software-lab.org