Analyzing Software using Deep Learning
Reasoning about Types and Code Changes with Hierarchical Networks (Part 2)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Summer 2020
Overview

- Hierarchical neural networks
- Type prediction
  Based on “TypeWriter: Neural Type Prediction with Search-based Validation” by Pradel et al., 2020
- Representing code changes
  Based on “CC2Vec: Distributed Representations of Code Changes” by Hoang et al., 2020
Types in Dynamic Progr. Langs.

- **Dynamically typed languages:**
  Extremely popular

- **Lack of type annotations:**
  - Type errors
  - Hard-to-understand APIs
  - Poor IDE support
Example

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"]
Gradual Typing

- Annotate **some code locations with types**
  - E.g., parameter types and return types of some functions only

- **Gradual type checker**
  - Warn about inconsistencies
  - Ignores missing information
Gradual Typing

- Annotate **some code locations with types**
  - E.g., parameter types and return types of some functions only
- **Gradual type checker**
  - Warn about inconsistencies
  - Ignores missing information

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
Overview of TypeWriter

Program → Lightweight static analysis → Neural type prediction

NL info → Probabilistic type prediction → PL info

Type vector → Static type checker ↔ Feedback-directed search

Search for consistent types → Program with type annotations
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
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"]
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"]
Output: Type Vector

- Type prediction as a classification problem

- Output of the model: Type vector
  - One element for each of top-1000 types
  - During training:
    All zero, except for the correct type
  - During prediction:
    Interpreted as probability distribution over types
Training the Model

- **Training data**: Existing type annotations
  - Multi-million line code base
  - Some types ($\approx 20$-$50\%$) already annotated
- **Learns to predict missing types from existing annotations**
Example of Predictions

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 of Predictions

```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:
- int, str, bool
- str, Optional[str], None
- List[str], List[Any], str
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
Overview of TypeWriter

Program → Lightweight static analysis → Neural type prediction

Program with type annotations → Feedback-directed search → Static type checker

Search for consistent types

NL info → PL info

Probabilistic type prediction

Type vector

Lightweight static analysis

Neural type prediction

Static type checker

Feedback-directed search

Search for consistent types
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:
    $$ (k + 1)^{|S|} \text{ possible type assignments} $$

Too large to explore exhaustively!
Feedback Function

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

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

- **Optimistic vs. pessimistic**
  - Add top-most predicted type everywhere and then remove types
  - Add one type at a time

- **Greedy vs. non-greedy**
  - If score decreases, keep the type
  - Backtrack to avoid local minima
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']
Results

■ **Neural model**

  □ Precision: 58-73% in top-1, 50-92% in top-5
  □ Recall: 50-58% in top-1, 69-72% in top-5

■ **Model and search together**

  □ Best strategy adds 72% of type-correct types and completely annotates 44% of files

■ **In use at Facebook**

  □ Thousands of suggested types accepted by developers with minimal changes