Type-guided synthesis for dynamic languages

Thomas Logan

1 Introduction

Dynamic programming languages can make writing programs quick and easy because they don't require specifying static bounds on behavior. Two of the most popular programming languages today, Javascript and Python, are dynamic programming languages. Javascript is the language of the web, while Python is the most popular choice for data science and machine learning projects. Consider a Python program that operates on strings or integers.

```python
def foo(x):
    if isinstance(x, int):
        return x + 1
    elif isinstance(x, str):
        return x + "..."
```

The dynamic language program avoids the extra step of associating terms with static bounds as seen in ML-like languages with the datatype mechanism.

```
datatype iors =
    Int of int |
    Str of string

fun foo(Int x) = x + 1
  | foo(Str x) = x ^ "..."
```

Although dynamic languages already offer ease and efficiency for writing programs, this facility can be enhanced with **synthesis of programs** from surrounding context. This article presents a system that synthesizes terms from context in a dynamic language. Synthesis of of programs for a dynamic language introduces a fundamental tension. While dynamic language programs benefit from a lack of

static bounds, program synthesis must be a terminating procedure driven by static bounds representing the goals of synthesis.

Types have become the lingua franca of formal specification of static bounds. Others have shown how various forms of specification, including examples, abstract values, pure propositions, and modal propositions, can be encoded as types. Types have been used successfully for verifying programs, guiding program synthesis in ML-family languages, and guiding humans in dependently-typed interactive theorem proving.

By **propagating** and decomposing types, it is possible to guide the synthesis of programs for dynamic languages. To maintain the spirit of dynamic languages, type annotations must remain optional. Previous work has demonstrated the utility of propagating types in theorem proving systems, local type checking, and synthesis of ML-family programs.

In order to efficiently guide program synthesis, types must be able to **express** fairly precise information. The system presented here offers types with expressivity comparable to typical decidable predicate logics.

Dynamically, two interfaces may have the same correctness result for some inputs, while differing on other inputs. In order to maintain some semblance of the dynamic style programming, while also adding safety with static behavior, **Subtyping** is used as the criteria for deciding if a term's composition is statically invalid.

In order to have fine-grained control over where types are utilized or avoided, types may be composed of **unknown types**, represented by "?" to indicate that any subterm corresponding to that portion of the type will not be subject to static constraints.

However, if type annotations are not available, then **types are inferred** from context. Taking advantage of type inference techniques from ML-family languages is a good starting point, but it's not sufficient. ML-family type inference is sound, because an ML term is intrinsically associated with a static bound in accordance with ML's datatype mechanism. Terms in dynamic languages are not intrinsically associated with a principal type. Type inference for dynamic languages cannot be sound without greatly violating the liberal reuse of terms in dynamic languages.

Thus, it is necessary to devise a method of type inference that is unsound, yet still provides sufficient information to guide efficient program synthesis. Additionally, despite being unsound, it should be able to prune/reject a significant portion of bad programs. It may reject good programs, but only if it can infer a type.

In parametric types where a generic input type must be the same as a generic output type, The dynamic nature of terms is at odds with inferring types. These generic types can be specialized based on the terms that are witnessed as inputs. Since dynamic languages cannot infer a principal type, The parameter type must accommodate types of unforeseen arguments, while the return type should be **widened** but restricted to previously arguments. This system tackles the tension in these goals by combining the union combinator with the unknown type.

In functions where a parameter has an unknown type and that parameter is internally used as an argument to an application, the dynamic nature of terms adds some complications to type inference. The argument type must avail itself to internal parameter types of unforeseen compositions, while the external parameter type should be **narrowed** and restricted to the internal parameter types of previously seen compositions.

2 Overview

3 Technical Details

4 Evaluation

5 Related work