Type-guided synthesis for dynamically typed languages

Thomas Logan

1 Introduction

Dynamically typed 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 dynamically typed. 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 + "abc"
```

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

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

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

Although dynamically typed 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 dynamically typed language. Synthesis of programs for a dynamically typed language introduces a fundamental tension. While dynamically typed 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. To maintain the spirit of dynamic types, 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. For example, the types are expressive enough to precisely specify a function that takes a natural number lstinlinen and an element lstinlinex, and returns list of n elements.

```
replicate :
∀ α . α → μ nat_to_list .
    *zero → *nil &
    ∀ nat list . (nat → list) ≤ nat_to_list .
        *succ nat → *cons (α × list)
```

Correspondingly, types can also specify a pair of a list and its length.

```
measurment :
∃ α . μ nat_and_list .
    *zero → *nil |
    ∃ nat list . (nat × list) ≤ nat_and_list .
        *succ nat × *cons (α × list)
```

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 dynamically typed programming, while also adding safety with static behavior, **subtyping** is used to decide 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 **dynamic**

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 dynamically typed languages are not intrinsically associated with a principal type. Type inference for dynamically typed languages cannot be sound without greatly violating the liberal reuse of terms afforded by dynamic types.

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 static 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.

```
(λ pair : ∀ α . α → α → (α × α) ⇒
(λ n : int ⇒ (λ s : str ⇒
  pair n s
)))
```

These generic types can be specialized based on the terms that are witnessed as inputs. Since a principal type cannot be inferred in dynamically typed languages, The parameter type must accommodate types of unforeseen arguments, while the return type should be **widened** but restricted to previously seen arguments. This system tackles the tension in these goals by combining the union combinator with the dynamic type.

Once `pair` is applied to an integer its generic type is specialized to `int` | ?, in which union with the dynamic type behaves like the top type ⊤, accommodating receiving a string as a the subsequent input. Thus, the generic type is specialized to `int` | `str` | ?. The result type for `pair n s` ends up as (`int` | `string`) × (`int` | `string`). The semantics of union with the dynamic type result in the dynamic type falling away for actual types.

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.

```
(λ i2n : int → nat ⇒
(λ s2n : str → nat ⇒
  λ x : ? ⇒ (i2n x, s2n x)
))
```

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. Once `i2n` is applied to `x`, the type for `x` is specialized to `int` & ?, in which intersection with the dynamic type behaves like the bottom type, availing itself to be used in subsequent applications. Thus, the dynamic type is inferred to be `int` & `str` & ?. The external parameter type for ends up as `int` & `str`. The semantics of intersection with the dynamic type result in the dynamic type falling away for expected types.

2   Overview

3   Technical Details

4   Evaluation

5   Related work