Type-guided synthesis for dynamically typed languages

Thomas Logan

ACM Reference Format:
Thomas Logan. 2022. Type-guided synthesis for dynamically typed languages. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

```
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-family 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.

Dynamically, two interfaces may have the same correctness result for some inputs, while differing on other inputs. **subtyping** is used to decide if a term's composition is statically invalid, maintaining semblance to dynamically typed programming, while also adding safety with static behavior.

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.

For example, in a program that extracts the first element of a pair of a particular type, it's possible to detect an error before knowing the second element.

to detect an error of a pairt of a

```
λ n : nat ⇒
  let first = (λ (x,y) : (str × str) ⇒ x) .
  first (n, _)
```

The applications' argument type (`nat × ?`) is propagated and decomposed such that the subtyping constraint `nat ⊆ str` is decided.

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.

If type annotations are not available, then **types are inferred** from context. The actual type of a term can be inferred in an obvious way.

```
#cons("hello",#cons("world",#nil())) :
#cons(str × #cons(str × #nil◊))
```


Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To infer expected types, type inference techniques for ML-family languages are a good starting point. 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 the principal type of any term in dynamically typed languages is top, 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.

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

```
let measurement : μ (nat × list) .
  #zero ◊ × #nil ◊ |
  #succ nat × #cons (? × list)
```

This is syntactic sugar for:

```
let measurement :
∃ α . μ nat_and_list .
  #zero ◊ × #nil ◊ |
  ∃ nat list ::
    (nat × list) ≤ nat_and_list .
    #succ nat × #cons (α × list)
```

Semantically, the type of measurement is similar to the definition of list and length in Synquid []

```
termination measure len :: List β → Nat
data List β where
  Nil :: {v: List β | len v = 0}
  Cons :: β → xs: List β →
    {v: List β | len v = len xs + 1}
```

Correspondingly, the types are expressive enough to precisely specify a function that takes a natural number `n` and an element, and returns a list of `n` elements.

```
let replicate :
∀ α . α → ν (nat → list) .
  #zero ◊ → #nil ◊ &
  #succ nat → #cons (α × list)
```

This is syntactic sugar for:

```
let replicate :
∀ α . α → ν nat_to_list .
  #zero ◊ → #nil ◊ &
  ∀ nat list ::
    nat_to_list ≤ (nat → list) .
    #succ nat → #cons (α × list)
```

The type of `replicate` is co-recursive rather than recursive. When expanded to the type with a universal type and constraint, the universally typed variables appear on the right hand side of the subtyping constraint, while the co-recursive variable appears on the left hand side. This is consistent with the notion that a intersection of types is a subtype of any one of its parts.

Semantically, the type of `replicate` is similar to the definition of `replicate` in Synquid []

```
replicate :: n:Nat → x:α →
  {v: List α | len v = n}
```

2 Overview

3 Technical Details

$$\textsc{var-left}\ \frac{\Delta(\alpha) = ? \qquad \text{fresh } \beta}{\Delta \vdash \alpha \leq \tau \rightsquigarrow \{\alpha \mapsto \tau \,\&\, \beta\}} \qquad \textsc{var-rght}\ \frac{\Delta(\alpha) = ? \qquad \text{fresh } \beta}{\Delta \vdash \tau \leq \alpha \rightsquigarrow \{\alpha \mapsto \tau \mid \beta\}}$$

$$\textsc{recur-base}\ \frac{}{\Delta \vdash \mu\alpha.\tau \leq \mu\alpha.\tau \rightsquigarrow \{\}} \qquad \textsc{recur-tag}\ \frac{\Delta \vdash \#l\ \tau_1 \leq \tau[\alpha/\mu\alpha.\tau] \rightsquigarrow \Delta_1}{\Delta \vdash \#l\ \tau_1 \leq \mu\alpha.\tau \rightsquigarrow \Delta_1}$$

$$\textsc{recur-record}\ \frac{\Delta\ ?\ \vdash_R (\text{linearize } \tau') \leq \mu\alpha.\tau \rightsquigarrow \Delta_1}{\Delta \vdash \tau' \leq \mu\alpha.\tau \rightsquigarrow \Delta_1}$$

$$\textsc{corec-base}\ \frac{}{\Delta \vdash \nu\alpha.\tau \leq \nu\alpha.\tau \rightsquigarrow \{\}}$$

$$\textsc{corec-case}\ \frac{\Delta \vdash \forall\beta :: \tau[\alpha/\nu\alpha.\tau] \leq (\beta \to \tau_2)\ .\ \beta \leq \tau_1 \rightsquigarrow \Delta_1 \qquad \Delta \vdash \forall\beta :: \tau[\alpha/\nu\alpha.\tau] \leq (\tau_1 \to \beta)\ .\ \beta \leq \tau_2 \rightsquigarrow \Delta_2}{\Delta \vdash \nu\alpha.\tau \leq \tau_1 \to \tau_2 \rightsquigarrow \Delta_1\Delta_2}$$

Figure 1. unification subtyping

4 Evaluation

5 Related work

Notes

contributions

1. program synthesis such that:
   a. goal defined by partial programs with missing annotations
   b. goal transformed into type specification
   c. type specification may contain dynamic type
2. algorithmic subsumption allows simple subtyping rule with dynamic type
   a. subtyping can have a dynamic type on either side without risk of all terms being accepted by typing rules (i.e. everything typing)
3. an expressive type system based on intersection, union, recursive, and co-recursive types
   a. without dependent types
   b. comparable to prolog
   c. allows dynamic type
   d. sound modulo absence of dynamic type or static type union/intersect with dynamic
   e. types behave either leniently or strict depending on usage as actual type or expected type
4. full type synthesis with full type propagation (for all rules)
5. type unification such that:
   a. the principal type of all terms is top
   b. lenient (but unsound) static type inference
   c. expanding type info with union and intersection and dynamic type
6. interleaving type constraint generation and solving with dynamic type

$$\textsc{record-unify-step}\ \frac{\Delta \vdash \tau_1 \leq \exists\beta :: (.l\ \tau_0\ \&\ \beta\ \&\ \tau_2) \leq \tau[\alpha/\mu\alpha.\tau]\ .\ \beta \rightsquigarrow \Delta_1 \qquad \Delta\ (\tau_0\ \&\ .l\ \tau_1) \vdash_R \tau_2 \leq \mu\alpha.\tau \rightsquigarrow \Delta_2}{\Delta\ \tau_0 \vdash_R .l\ \tau_1\ \&\ \tau_2 \leq \mu\alpha.\tau \rightsquigarrow \Delta_1\Delta_2}$$

$$\textsc{record-unify-base}\ \frac{\Delta \vdash \tau_1 \leq \exists\beta :: (.l\ \tau_0\ \&\ \beta) \leq \tau[\alpha/\mu\alpha.\tau]\ .\ \beta \rightsquigarrow \Delta_1}{\Delta\ \tau_0 \vdash_R .l\ \tau_1\ \leq \mu\alpha.\tau \rightsquigarrow \Delta_1}$$

Figure 2. record unification subtyping

differences

from gradual typing:

1. infers more static type info
2. stronger soundness claim (i.e. weaker soundness pre-condition)
3. dynamic semantics are irrelevant
4. deterministic subsumption
5. dynamic type built into subtyping

from HM type inference:

1. synthesizes type in addition to constraint solution due to subtyping
2. propagates types
3. global top type, leaves types open during unification
4. expands and narrows types

from Roundtrip typing:

1. type combinators without dependent types
2. prolog-like type language

3. synthesizes types for all rules
4. global top type, leaves types open during unification
5. expands and narrows types
6. delegates to unification to decompose types
7. delegates to unification to free bound variables

gradual typing

How gradually typed inference typically works:

1. infer either a strict static type or a dynamic type
2. let annotations implicitly guide the usage of static vs dynamic checks

There exists work on type inference for gradually typed programs. To the best of my knowledge, they are all sound and complete. Other gradual type inference systems separate generating constraints from solving constraints. Gradual typing systems use the dynamic type `(?)` to indicate that dynamic checks should be used. Gradual type inference relies on ML types with union extension. They do not contain the idea of using intersection and union to keep types open. Gradual typing systems separate subtyping from the relation between dynamic to static types to avoid all terms typing via the subsumption rule.

references

1. Refinement types for ML by Tim Freeman and Frank Pfenning
2. Refinement types as proof irrelevance by William Lovas and Frank Pfenning
3. Local type inference by Benjamin C. Pierce and David N. Turner
4. Liquid types - http://goto.ucsd.edu/~rjhala/papers/liquid_types.pdf
5. Program synthesis from polymorphic refinement types by Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama
6. Example-directed synthesis: a type-theoretic interpretation by Frankle, Osera, Walker, and Zdancewic
7. Gradual typing for functional languages by Jeremy G. Siek and Walid Taha
8. Gradual typing for functional languages by Jeremy G. Siek and Walid Taha
9. Gradual typing for objects by Jeremy G. Siek and Walid Taha
10. Gradual typing with unification-based inference by Siek and Manish Vachharajani
11. Dynamic Type Inference for Gradual Hindley–Milner Typing - https://dl.acm.org/doi/pdf/10.1145/3290331
12. Principal type schemes for gradual programs - https://www.cs.ubc.ca/~rxg/ptsgp.pdf
13. Gradual Liquid Type Inference - https://arxiv.org/pdf/1807.02132.pdf
14. Gradual typing with union and intersection types - https://dl.acm.org/doi/pdf/10.1145/3110285
15. Gradual typing: a new perspective - unions, intersections, dynamic type, and type inference - https://www.irif.fr/~gc/papers/popl19.pdf
16. Gradual refinement types (via abstract interpretation) - https://pleiad.cl/papers/2017/lehmannTanter-popl2017.pdf
17. Consistent subtyping for all - https://xnning.github.io/papers/consistent-subtyping-for-all-toplas.pdf
18. Polymorphic functions with set-theoretic types, part 1 - https://www.irif.fr/~gc/papers/polydeuces-part1.pdf
19. Polymorphic functions with set-theoretic types, part 2 - https://www.irif.fr/~gc/papers/polydeuces-part2.pdf
20. Revisiting occurrence typing - https://arxiv.org/pdf/1907.05590.pdf