Automatic Type Migration Using Symbolic Evaluation and SMT Solvers

Andrew Li, Bennett Lindberg, Christos Dimoulas
Department of Computer Science, Northwestern University, Evanston, IL, USA

Abstract

Software engineers employ both dynamically- and statically- typed languages in software development. Dynamically typed programs are easier and faster to prototype, while statically typed programs are type safe and easier to maintain and document [1]. A combination of both dynamic and static typing would improve software development by combining the best of both worlds. Gradually adding type annotations to dynamically typed languages is a slow and time-intensive process that is prone to errors. Existing work on the gradual typing problem is either focused on scalable but imprecise, or precise but non-scalable solutions. These solutions are limited. We demonstrate a prototype to precise type annotation for larger-scale programs using symbolic evaluation and SMT solvers [3,4]. Types are modelled as trees with interior nodes representing concrete compound types and leaves representing symbolic primitive types. Our approach deduces types by observing usage at run-time. Types of leaf nodes are generated by symbolic constraint generation and integration with an SMT solver. Our implementation generates precise types for small, non-recursive programs that use integer, boolean, struct, and function types along with integer operations, conditionals, variable assignments, struct operations, and higher-order functions. The work here demonstrates the feasibility of using symbolic evaluation to annotate types.

Background

Mainstream programming languages are statically or dynamically typed:

A
```
int addOne(int x) {
    return x + 1;
}
addOne("banana") // fails at compile time
```

B
```
def add_one(x):
    return x + 1
add_one("banana") # fails at run time
```

C
```
(: add-one : Integer -> Integer)
(define (add-one x)
    (+ x 1))
(add-one "banana") ; fails at *compile* time
```

D
```
(define (add-one x)
    (+ x 1))
(add-one "banana") ; fails at *run* time
```

Figure 1: (**A-B**) statically and dynamically typed code written in commonly used languages; (**C-D**) the same code written in Racket, the language used to create our prototype.

Statically typed languages are safer and easier to maintain, as the code is often self documenting [1].

Dynamically typed languages are popular among developers because they are more flexible and code takes less time to write.

Automatic migration from dynamic to static typing would let developers quickly expand their projects with dynamically-typed code without sacrificing project maintainability and type safety.

Existing work is limited, either in scale or precision. Prior works targeting real-world programming languages often fall back on run-time type checks, while approaches that generate precise types generally work only on a few language features.

`Order3-Fun` Benchmark: `fun(f : ¤).fun(x : ¤).x (f x)`

```
(define (foo f x)
    (x (f x)))
```

Types of `f` and `x`?

A **Literature ("the best so far")**
fun f : (α → β) → α
fun x : α → β

B **Our Implementation**
fun f : (any → any) → any
fun x : any → any

C **Large Language Model (o3-mini-high)**
fun f : (α → β) → α
fun x : α → β

Figure 2: Results of literature implementation [2], our implementation, and a language model implementation on the `Order3-Fun` benchmark. Note that our implementation does not support having multiple generic types like the α and β shown above, while a language model is able to.

Northwestern | McCORMICK SCHOOL OF ENGINEERING
Computer Science

Methodology

We generate types using symbolic evaluation and SMT solvers.

Symbolic Evaluation: An evaluation technique where values are represented using placeholder symbols and reasoned about using theoretical principles. Symbolic evaluation is useful when reasoning about infinitely-sized concrete value spaces [3].

```
(define (foo bar)
  (define function
    (lambda x (+ x 1)))
  (function bar))
```
`(foo 5)`
`; we have a lambda that adds 1`
`; (function 5) = (+ 5 1) = 6`
What about other integers like 9?

`(foo (symbolic-integer))`
`; : function : Integer -> Integer`
`; (function (symbolic-integer)) = Integer`
Integer output for *all* Integer inputs

Figure 3: Symbolic evaluation on a simple function, which demonstrates that evaluating on symbolic inputs can help us infer more type information than evaluating on concrete inputs.

SMT: Satisfiability Modulo Theories, A set of problems where one seeks to satisfy mathematical formulas by assigning concrete values to variables. SMT supports various data types by applying their respective theories [3, 4].

A
```
(+ x 5) ; x must be Integer -- (assert-type x Integer)
(if y 1 x) ; y must be Boolean -- (assert-type y Boolean)
```
→ `x: Integer` `y: Boolean`

B
```
(+ x 5) ; x must be Integer -- (assert-type x Integer)
(if x #t #f) ; x must also be Boolean -- (assert-type x Boolean)
```
→ `(unsat)`

Figure 4: Use of SMT assertions to deduce types. (**A**) Where there exists a solution, we generate the corresponding types of variables. (**B**) Where there are contradictions, the type is "unsat" (unsatisfied).

During program evaluation, we start by assuming a value can be any type. We represent values as a union of an Integer type and a Boolean type — a tree. Structs are represented as deeper trees which can expand.

Symbolically running code line by line gives assertions for the trees which allows for us to collapse trees into specific types. A similar process works for higher-order functions.

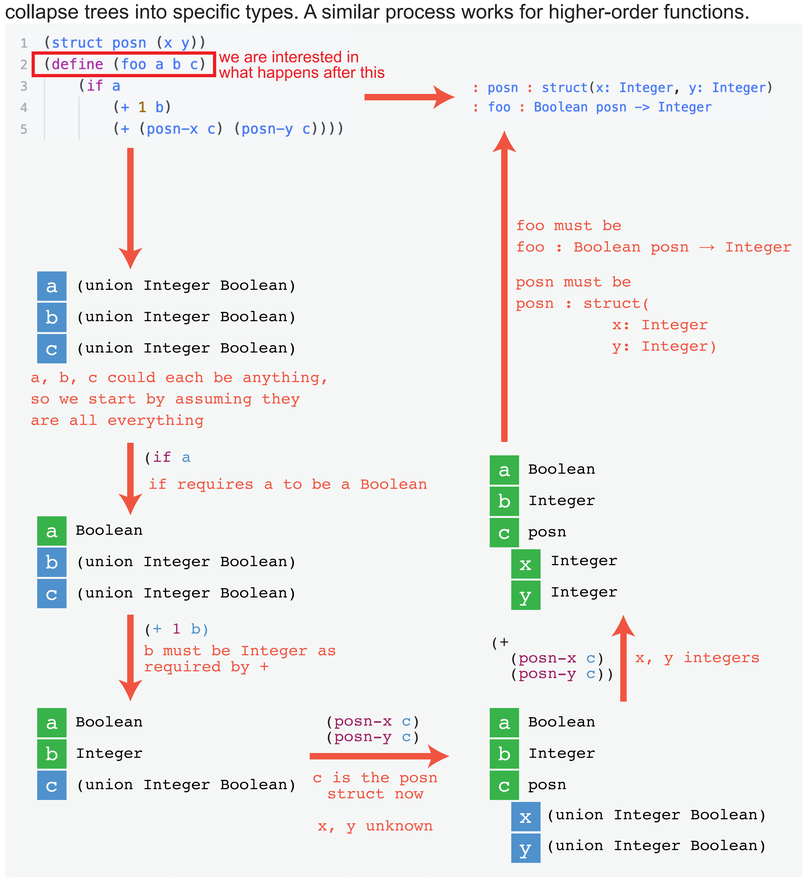

Figure 5: Diagram of symbolic execution of a simple Racket program, showing how individual lines of code yield specific type assertions, and how structs lead to tree expansion.

Outcomes

We implemented our approach in Racket/Rosette. We currently support structs, higher-order functions, and integer and boolean primitive types.

A
```
(struct posn (x y))
(define (foo a b)
   ((posn-x a) (b 1 #f)))
```
```
foo : ((Structof [x : (any -> any)]
                 [y : any]))
      (integer boolean -> any)
      -> any)
```

B
```
(define (sift n st)
   (define hd (stream-first st))
   (define tl ((stream-rest) st)))
   (cond [(= 0 (modulo hd n)) (sift n tl)]
         [else (make-stream hd (lambda () (sift n tl)))]))
```

C
```
(any (Structof [first : integer]
               [rest : (-> any)])
   -> any)
```

D
```
(integer (Structof [first : integer]
                   [rest : (-> any)])
         -> (Structof [first : integer]
                      [rest : (-> any)]))
```

Figure 6: Results of our implementation (**A**) on a program with nested struct input types and if branches, and (**B**) on the sieve gradual typing benchmark. Our implementation (**C**) struggles with recursion, but is able to add types to much of the benchmark compared to the ground truth (**D**).

Our approach was tested on the prime sieve benchmark [5]. Our solution identifies the majority of the type tree structure of the program. Our implementation accurately deduces several primitive types as integers, but most primitive types remain unknown.

Our solution cannot generate types for recursive functions, which limits the real-world scalability of our work.

Conclusions and Future Work

Our prototype is a proof of concept for using symbolic evaluation and SMT solvers in generating types.

Our implementation avoids needing concrete inputs; we treated symbolic types as inputs and explore all paths of a program.

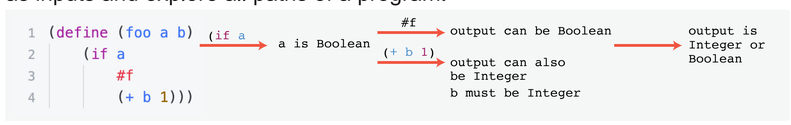

Figure 7: Path execution of a program with multiple paths. Both branches of the `if` block are used to deduce the type of the output, which here is a union of both `Integer` *and* `Boolean`.

Our implementation currently does *not* natively support union types, like the example shown in Figure 7. Supporting union types is a direction for future work. Our work also does not support recursion, like the code shown in Figure 8.

A future direction may be to integrate a language model. Large language models with a good prompt can better pattern match than our implementation, even for niche or self-dependent / cyclic patterns such as recursion.

```
(define (fibonacci n)
   (if (or (= n 1) (= n 2))
       1
       (+  (fibonacci (- n 1))
           (fibonacci (- n 2)))))
```
`; reasoned for 5 seconds`
`fibonacci: (Integer -> Integer)`
`    n [fibonacci]: Integer`

Figure 8: Large language model type annotation of a recursive program in 5 seconds of reasoning, on a program that our tree-based implementation cannot annotate. Using a language model's annotations as a starting point may lead to more precise types, and scale to more complex features.

References & Acknowledgements

[1] S. Tobin-Hochstadt, M. Felleisen, R. Findler, M. Flatt, B. Greenman, A. M. Kent, V. St-Amour, T. S. Strickland, and A. Takikawa, "Migratory typing: Ten years later," in *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA* (B. S. Lerner, R. Bodík, and S. Krishnamurthi, eds.), vol. 71 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[2] Jeremy G. Siek and Manish Vachharajani. 2008. Gradual typing with unification-based inference. In Proceedings of the 2008 symposium on Dynamic languages (DLS '08). Association for Computing Machinery, New York, NY, USA, Article 7, 1–12. https://doi.org/10.1145/1408681.1408688

[3] P. C. Nguyen, S. Tobin-Hochstadt, and D. Van Horn, "Soft contract verification," *ACM SIGPLAN Notices*, vol. 49, no. 9, pp. 139–152, 2014.

[4] L. Phipps-Costin, C. J. Anderson, M. Greenberg, and A. Guha, "Solver-based gradual type migration," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[5] B. Greenman, "GTP benchmarks for gradual typing performance," in *Proceedings of the 2023 ACM Conference on Reproducibility and Replicability*, ACM REP '23, (New York, NY, USA), p. 102–114, Association for Computing Machinery, 2023.