Symbolic Generation of Precise and Scalable Types

Li, Andrew; Lindberg, Bennett

December 9, 2024

1 Abstract

Programmers often use both statically- and dynamically-typed languages in software development. Although developers prefer the flexibility of dynamically-typed languages [1], statically-typed languages offer safer compile-time type checking and faster code execution [2, 3]. However, manually annotating untyped programs to benefit from static typing is prohibitively expensive. Developers could immensely benefit from automated type migration of their untyped code bases.

Many existing type migration solutions target real-world software with complex language features that are difficult to analyze statically [4]. These large-scale solutions often fall back on slow run-time type checks where static type inference is difficult [4]. Other existing solutions statically generate more precise type annotations but target much smaller-scale languages with fewer real-world language features. We hypothesize that generating type annotations through a mixture of symbolic execution [5], constraint generation [6], and SMT solvers [6] will better optimize for both precision and scalability than existing literature.

2 Research Context and Problem Statement

Mainstream programming languages are generally either statically- or dynamically-typed. Statically-typed languages enforce type soundness at compile time and often require explicit type declarations. Dynamically-typed languages are more flexible; they allow types to change at run time and let programmers omit type declarations [7]. Researchers claim dynamic typing is both easier to use [8] and more flexible and expressive [2, 8, 9]. Accordingly, many developers choose dynamically-typed languages—such as Perl, Python, Ruby, and JavaScript—for software development projects to increase productivity [1].

Developers who choose dynamically-typed languages eventually encounter significant downsides [1]. Developers find large, dynamically-typed projects difficult to maintain because they lack type annotations as documentation [1]. Moreover, dynamic typing reduces type safety by delaying type errors to run time [2, 5, 3] and inhibits type-related compiler optimizations [3, 1]. Developers can alleviate these downsides by migrating to static typing, but this requires writing type annotations, which is time intensive [6]. Large code bases, such as the 500,000-line untyped pedagogical Racket code base [1], are impractical to manually annotate with types. Automated generation of types for untyped projects would alleviate the time cost of type migration, yielding developers the benefits of statically-typed code while maintaining the ease of writing dynamically-typed code.

Useful automatic type migration systems must generate precise (i.e. accurate and specific) types to provide quality static error checking, and they must scale to support complex language features that apply to large code bases. However, type migration researchers have generally failed to achieve both precision and scalability. Phipps-Costin et. al. [6] provide precise gradual type migration, but restrict their work to the small-scale Gradually Typed Lambda Calculus. Furr et. al. [4] annotate untyped Ruby code and scale their work to real-world Ruby programs, but they fall back on slow, dynamic type checks where static types cannot be inferred. This

research project seeks to investigate the precision-scalability trade-off by answering the following research question: how can precise type annotations be generated for dynamic and untyped programs while maintaining scalability?

We plan to leverage symbolic evaluation, constraint generation, and SMT solvers to generate precise and scalable types for untyped Racket programs. We will extend Nguyen et. al.'s work [5] on verifying contracts with symbolic evaluation to represent unknown types as symbolic values. Then, we will automate program traversal to generate constraints on each symbol's type, similar to Phipps-Costin et. al.'s construct-specific type constraint generation [6]. Last, we will follow Phipps-Costin et. al. and solve these constraints using an SMT solver to produce concrete type annotations for each type symbol.

3 Related Work

Current approaches to automated type migration are limited in either scale or precision. The TypeWhich migration tool presented by Phipps-Costin et. al. [6] yields type annotations that are generally more precise than other tools targeting the same language constructs. The scale of the TypeWhich project is restricted to the Gradually Typed Lambda Calculus, a small-scale language lacking common language patterns (such as linked-lists, while-loops, structs, etc.) found in real-world, larger-scale code bases.

Other approaches support more diverse language features in order to target larger code bases but fall back to dynamic type checking where types cannot be statically inferred. Furr et. al.'s profiling tool, $\mathcal{P}$Ruby, supports static type inference in Ruby (a typically dynamically-typed language) [4]. $\mathcal{P}$Ruby handles difficult-to-analyze language features such as `eval`, which interprets raw input strings as Ruby code [4]. Dynamic features like `eval`, due to their arbitrary run-time behavior, increase the difficulty of generating type annotations. Furr et. al. employ high-level code profiles to understand the behavior of these dynamic features during run time, allowing for precise retroactive type annotation [4]. The $\mathcal{P}$Ruby project is unable to fully migrate untyped Ruby programs, so Furr et. al. employed dynamic type checks where types cannot be determined from the profiles [4].

In addition to code profiling, $\mathcal{P}$Ruby and other Ruby-related works use their input program's existing test cases to generate type annotations [4, 10]. The use of test cases is similar to code profiles, with both techniques employing existing code to understand a program's run-time behavior. Although programmers are arguably burdened by writing test cases, Furr et. al. reason that large-scale programs should already have a representative set of unit test suites [4].

Other related approaches perform static type checking instead of migration. Ren and Foster's Hummingbird tool [11] statically type checks methods at run time on a per-invocation basis, considering the types of the particular invocation's arguments. Hummingbird's run-time nature enables type checking on all method applications that occur during program execution, removing the need to consider all possible scenarios at compile time. However, this run-time type checking generates overhead, considerably slowing down program execution [11]. In contrast, Nguyen et. al.'s *soft contract verification* system [5] performs static checks entirely at compile time, removing run-time overhead. Nguyen et. al. cover all possible run-time scenarios at compile time by reasoning about the properties of abstract values instead of testing all concrete examples. However, the *soft contract verification* system targets provided software contracts (not types) and does not infer types [5].

Limitations in current work reduce to limitations in scale or limitations in precision. Some existing projects explore certain language features or other ways to increase the efficiency of type annotation. Other existing projects systematically generate type annotations but for languages without these features. A next step in this field is to combine existing work to develop a scalable but still precise type-annotation solution.

4 Proposed Solution

This research will combine several techniques from previous literature in type migration and inference to build an exact and scalable automatic type migration solution. In particular, our solution will generate types using symbolic evaluation, construct-specific type constraint generation, and SMT solvers. Our approach begins by modeling unknown type annotations as symbolic constants. We will walk the program code and precisely consider encountered language constructs to obtain type constraints on these symbols. Then, we will use an SMT solver to reconcile the constraints and produce a valid typing of the program.

We will target the migration of untyped Racket programs into semantically-equivalent Typed Racket programs. Typed Racket is the statically-typed sister language of Racket. We will utilize Rosette, a Racket-based language, to streamline the type migration process. Rosette provides constructs for creating symbolic constants, specifying constraints, and solving constraints. Rosette also contains infrastructure for symbolic reasoning during program execution.

4.1 Symbolizing Types

We base our type deduction logic on symbolic reasoning about types. Researchers use symbolic evaluation to reason abstractly about placeholder symbols using theoretical principles [5]. Consider the following function, where `x` is always a positive integer and `y` is always a negative integer:

```
(define (mult x y) (* x y))
```

We can reason that `mult` always returns a negative integer based on the mathematical principle that the product of a positive and negative integer is always a negative integer [5]. Symbolic reasoning lies in opposition to concrete evaluation, which involves reasoning directly about specific values. To reason concretely about `mult`, we would provide example positive and negative integers to the function and observe the results.

Symbolic verification of software contracts by Nguyen et. al. [5] inspired our decision to use symbolic evaluation. Like types, software contracts capture and constrain properties about values. The authors represent values as abstract symbols and collect properties about the symbols as they flow through input programs in order to check contract satisfaction [5]. Nguyen et. al. use symbolic reasoning to make contract verification tractable; concrete testing would require them to verify contracts on the (potentially infinite) set of all valid concrete inputs. For the same reason of tractability, we will need symbolic evaluation to achieve large-scale type migration. Where Nguyen et. al. represent contracted values as symbols, we will represent type annotations as symbols. In the first stage of our solution, we will replace each unknown type annotation of an input program with a symbolic constant. We will later constrain these type-representative symbols, similar to how software contracts in [5] narrow value-representative symbols.

We will walk the input program to replace unknown type annotations with symbols. During our traversal, we will apply construct-specific reasoning to identify missing type annotations. As an example, take the following function:

```
(define (foo x y) ...)
```

We may reason that a type annotation on `foo` is missing, necessitating the creation of three symbolic constants: one for `x`, one for `y`, and one for `foo`'s return value. We will use Rosette's `define-symbolic` function to create these symbolic constants as first-class values that can be passed around and constrained. Each symbolic constant will also be added to a global symbol table to enable future reference within constraints.

4.1.1 Handling Unsolvable Types

We will use symbolic evaluation to reason about a wide range of types prevalent in real-world code bases. However, Rosette limits symbolic reasoning to `integer` and `boolean` types to ensure SMT solvers can reliably solve its constraints. To enable symbolic evaluation of complex data types (such as `struct` and `list` types) that Rosette considers unsolvable, we must encode symbolic equivalents of these types using `integer` and `boolean` symbols.

Fortunately, Rosette's symbolic unions work well for this purpose. Symbolic unions are variables with underlying values guarded by one or more symbolic booleans. For example, we can define a simple symbolic union `u` as:

```
(define-symbolic guard boolean?)
(define u (if guard a b))
```

Since the value of `u` is guarded by a symbolic boolean, `u` is effectively a symbolic constant representing values in the set $\{\texttt{a}, \texttt{b}\}$. By nesting boolean guards, we can expand the set of values `u` can represent. In turn, we can define symbolic constants representative of custom concrete value sets (not just the set of all integers or the set of all booleans).

To create symbolic constants for unsolvable types, we must encode each type's concrete value set as a symbolic union. We can view symbolic unions as ordered sets of symbolic booleans that map all permutations of `true` and `false` to another set of values:

$$\texttt{boolean-permutations} \longleftrightarrow \texttt{concrete-values}$$

To encode unsolvable types, we will devise mappings between each type's concrete values and boolean permutations. Then, we will encode the mappings as symbolic unions with nested boolean guards, thereby producing symbolic constants for each type.

4.2 Constraining Symbolic Types

We must carefully reason about the underlying program to accurately constrain symbols. Our approach involves identifying reliable type constraint generation rules specific to common language constructs, similar to the type constraint generation performed by Phipps-Costin et. al. [6]. Phipps-Costin et. al. traverse input programs in their entirety, collecting type constraints from language constructs as they encounter them. For example, given $a \times b$ where multiplication is defined only on integers, Phipps-Costin et. al. collect three new type constraints: a is an integer, b is an integer, and $a \times b$ is an integer (assuming the program is well-typed). The authors establish similar constraint generation patterns for other common constructs, such as function applications [6].

We take a similar approach to constraint generation as [6], although we seek to extend the logic to a broader set of language constructs to maximize scalability. Moreover, instead of constraining values, we will assert about type annotations. In particular, we will confine the type-representative symbolic constants we create during our program walk. Referencing our global symbol table, we will produce new assertions on the relevant symbols as we encounter language constructs. We will leverage Rosette's `assert` function and first-class treatment of symbolic constants to make assertions on the symbols, which Rosette automatically collects into a separate assertion table.

Ultimately, we need to closely consider the constraint generation rules for each supported construct. We must develop constraint generation logic for a wide variety of language constructs (scalability) while maintaining the correctness and specificity of each rule (precision) to obtain a successful solution.

4.3 Generating Concrete Types

To produce a valid typing of an input program, we will reconcile our type constraints. In particular, we require a mapping of our type-representative symbols to concrete types that satisfies all of our constraints. We will leverage SMT solvers to find this mapping. SMT solvers algorithmically identify assignments of concrete values to variables that satisfy a body of constraints [5, 6]. Researchers enhance SMT solvers by adding theories, which provide sets of mathematical principles within which the solvers can operate. For example, an SMT solver with the theory of integers can reason about relational comparisons between integers like $\geq$ or $<$ [5].

We will use an SMT solver equipped with the theory of integers to reason about our type constraints. Our generated type constraints will be equality assertions between types, such as the following:

```
(assert (= type-of-a String))
(assert (= type-of-b type-of-a))
```

However, to use SMT solvers directly with non-standard data types, we need to create custom embeddings [5]. Instead of writing a new embedding for

type equality (as Phipps-Costin et. al. did in [6]), we will maintain a mapping between types and integers. We will then use Rosette's `solve` function to apply the Z3 SMT solver to *integer* equality assertions. Z3 will yield a mapping of symbolic constants to integer values, which we can then map back to concrete types.

5 Evaluation and Implementation Plan

Our ability to strike an optimal balance between precision in type constraint generation and scalability to real-world programs will determine our solution's performance. When implementing our solution, we will develop accurate type generation rules for the most common language features found in real-world programs, followed by expansion to less common constructs.

5.1 Evaluation Plan

We will measure our solution's performance in two separate areas: scalability and precision. A successful solution will have both; generating precise types is necessary for the migrated program to provide useful static error-checking and scaling up to large programs is necessary for real-world applicability.

We must consider the solution's scalability in both size and complexity of the input program. Real-world programs are large, potentially spanning many thousands of lines of code and multiple modules. They are also intricate, often containing a mix of both common and rarer language constructs. For both size and complexity, we will evaluate our solution's scale by comparing it with predefined levels. We will generate a list of target language features sorted by prevalence and decide what number of lines of code, modules, or functions constitutes a small, medium, and large-sized program. Then, we will determine our solution's performance level by observation (in the case of features) and experimentation (in the case of size). A more successful solution will support more language features and larger input programs.

We will evaluate precision by comparing the quality of the types generated by previous solutions in the literature against those generated by our own solution. Prior work on type migration provides numerous difficult-to-type benchmarking examples [6, 2, 8]. We plan to evaluate our solution against these benchmarks and consider the precision of the resulting types. We will evaluate precision in terms of both correctness and specificity. To understand how precision can vary, consider the generated type for `x` in the following example function adapted from [6]:

```
(define (foo x y) (y (x y)))
```

The most precise type for `x` is `(any -> any) -> any` because `x` is applied as a function to `y`, and `y` itself is used as a function. A type of `any -> any` is unspecific because `x` is applied to a function argument. A type of `(int -> int) -> any` is incorrect because no information supports a restriction to integers. A successful solution will generate the most specific types possible on these benchmarks without constricting the program's semantics.

5.2 Timeline

We began this research during the Fall 2024 quarter and will continue it through the Winter 2025 quarter. We plan to complete the work in three primary rounds, the first of which we have largely completed.

First, we will conduct a familiarization round to build the foundation for our future work. Thus far, we have gained a thorough understanding of Rosette and how to apply its relevant tools to type migration. We have also experimented with manual symbolic constant and constraint generation directly within Rosette. We will spend the remainder of the Fall 2024 quarter optimizing the integration of our implementation with Rosette's built-in symbolic evaluation functionality.

Given our work thus far, we are well positioned to begin automating symbol and constraint generation in Rosette during the Winter 2025 quarter. We will use the constraint rules we identified during manual experimentation to develop infrastructure that automatically locates missing type annotations and generates constraints on those annotations. We will

start by automating constraint generation for commonly used language constructs such as conditionals and struct methods. Before moving to more complex constructs, we will evaluate our solution on applicable code examples for precision and refine our technique as necessary. We expect to dedicate the first four weeks of the quarter to this process.

Our final phase will consist of expanding our previous work to more complex language constructs. We will iteratively identify and automate constraint rules for new constructs and test the modified solution for precision. After adding support for many constructs, we will evaluate the scalability of the solution, ideally reaching the ability to type common real-world programs of medium size. We expect to spend approximately four more weeks of the Winter 2025 quarter iterating on and evaluating the solution. We reserve the remaining two weeks of the quarter for further solution optimization and the potential to incorporate additional type migration techniques.

References

[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] A. Aiken, E. L. Wimmers, and T. Lakshman, "Soft typing with conditional types," in *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 163–173, 1994.

[3] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, "Dynamic typing in a statically typed language," *ACM Trans. Program. Lang. Syst.*, vol. 13, p. 237–268, Apr. 1991.

[4] M. Furr, J.-h. An, and J. S. Foster, "Profile-guided static typing for dynamic scripting languages," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pp. 283–300, 2009.

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

[6] 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.

[7] L. Tratt, "Dynamically typed languages," *Advances in Computers*, vol. 77, pp. 149–184, 2009.

[8] R. Cartwright and M. Fagan, "Soft typing," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pp. 278–292, 1991.

[9] R. Chugh, P. M. Rondon, and R. Jhala, "Nested refinements: a logic for duck typing," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 231–244, 2012.

[10] J.-h. An, A. Chaudhuri, J. S. Foster, and M. Hicks, "Dynamic inference of static types for ruby," *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 459–472, 2011.

[11] B. M. Ren and J. S. Foster, "Just-in-time static type checking for dynamic languages," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 462–476, 2016.