Towards Practical Verification of Dynamically Typed Programs

Björn Engelmann

Department of Computing Science, University of Oldenburg, Germany
bjoern.engelmann@uni-oldenburg.de

Dynamically typed languages enable programmers to write elegant, reusable and extendable programs. Recently, some progress has been made regarding their verification [2]. However, the user is currently required to manually show the absence of type errors, a tedious task usually involving large amounts of repetitive work.

As most dynamically typed programs only occasionally divert from what would also be possible in statically typed languages, properly designed type inference algorithms should be able to supply the missing type information in most cases [1].

We propose integrating a certified type inference algorithm into an interactive verification environment in order to
a) provide a layer of abstraction, allowing the users to verify their programs like in a statically typed language whenever possible and
b) use verification results to improve type inference and thus allow type checking of difficult cases.

As this is work in progress, we will in the following present the basic idea of our approach rather informally on the basis of small examples.

Model Language. For our investigation we use a minimalistic dynamically typed model language called *dyn* allowing us to focus on the problems arising from dynamic typing alone.

Dyn is imperative and object-oriented. It distinguishes syntactically between local variables (`v`) and instance variables (`@v`). The remaining syntax should be intuitively understandable.

Two built-in functions for checking object identity (`=`) and the runtime class of an object (`is_a`) are also provided. Primitives like numbers, strings and lists can be defined within *dyn*.

Type Inference. The goal of our type system is to statically prevent calling unsupported methods. As usual, the set of type error free programs is under-approximated as an exact solution is undecidable. We will later discuss some examples of correct programs that cannot be typed in our system.

Our type inference algorithm is based on [3], however the presentation focuses on what is necessary for the purpose of this writing. (* = Kleene star)

$$U ::= \{Classname^*\} \quad V ::= [\![S]\!] \quad T ::= V|U \quad C ::= T \subseteq T$$

The type language contains only union types (U) which denote all instances of the named classes. To infer these types for a given program P, we first assign each subexpression S a type variable $[\![S]\!]$ and then relate these variables with set inclusion constraints (C) generated by recursively traversing the parse tree of P.

This work is supported by the German Research Foundation through the Research Training Group (DFG GRK 1765) SCARE (www.scare.uni-oldenburg.de).

Oxhøj et al. [3] give a method for solving such constraint systems and properly handling dynamically dispatched method calls. In the end, we will either find the constraints to be inconsistent or have a proper solution σ assigning a union type $\sigma([\![S]\!])$ to every subexpression S of P.

Assertion Language. Our assertion language (P) contains the usual connectives from predicate logic and separation logic [4] along with typing assertions[2]. ($^+$ = Kleene plus)

$$\begin{array}{lll} P ::= & P \wedge P | P \vee P | \neg P | P \rightarrow P | (P) | \forall Id^+.P | \exists Id^+.P & T' ::= U | [\![E]\!] | [\![E.\texttt{@}Id]\!] \\ & | \texttt{emp} | P * P | P \mathrel{-\!\!*} P | E.@Id \mapsto E | E | T' \subseteq T' | T' = T' & \\ E ::= & \texttt{null} | Id | \texttt{@}Id | \texttt{self} | E.Id\texttt{(}E^*\texttt{)} | \texttt{new } Id\texttt{(}E^*\texttt{)} | \texttt{if } E \texttt{ then } E \texttt{ else } E \texttt{ end} | \texttt{result} & \end{array}$$

A subset of *dyn* expressions (E) can also be used in assertions[3]. The assertion language can thus be extended by the user as needed. However, method- and constructor calls are restricted to those proven to be total (terminate on all inputs) and side-effect-free. Contrary to programs, only well-typed expressions (inferable by our type inference algorithm) are allowed in assertions. In postconditions, `result` denotes the return value of the expression.

Examples. For most *dyn* programs (a polymorphic version of) the given type inference algorithm should be able to automatically infer the missing type information and thus allow reasoning like in a statically typed language. In the following, we will give some typical examples of non-typable programs and demonstrate how correctness can still be established.

Dynamic Type Check. In example ① the expression `self.howlong(3)` is not typable (class `numeric` does not support a method `length()`) although it can be executed without problems. The type system is not control flow sensitive and does not understand dynamic type checks (`is_a`). However, adding the assertion in line 3 corrects this problem. Assertions are control-flow sensitive as they denote properties of program states whose control flow reaches that point. The type expression $[\![\texttt{s}]\!]$ in the assertion thus denotes the type the parameter `s` has at that point rather than its general type. When encountering such an assertion, the constraint generation will introduce a new type variable $[\![\texttt{s}]\!]' = [\![\texttt{s}]\!] \cap \texttt{\{string\}}$ that is guaranteed to satisfy the given constraint and use it as the type of `s` until this assumption is invalidated (i.e. due to an assignment to `s` or a control flow join).

Note that such "type filters" need to be proven (like any other assertion) in order to preserve the type system's soundness. In this example, the proof is straightforward due to the postcondition of `is_a` and the semantics of the if statement.

Imprecise Control Flow Abstraction. In example ② we assume that numerics can be added to numerics and strings can be added (concatenated) with strings but there is no addition defined for combinations of the two (such attempts thus yield a type error). In this scenario, both `x` and `y` would be given the type `{numeric, string}` and example ② would not be typable as the algorithm would suspect that a `numeric` could be added to a `string` in line 10. Since the control flow is joined after the if statement, making the algorithm control flow sensitive does not help either.

Adding the assertion in line 9 solves this problem, since the expression `x + y` is well-typed under both possibilities. Establishing it is also simple as it holds at the end of both branches

[2] $[\![E.@Id]\!]$ denotes the type of the instance variable $@Id$ of the object referenced by E

[3] $E_1.@Id \mapsto E_2$ requires the instance variable $@Id$ of E_1 to point to E_2 and E is a shorthand for $E == true$

Examples:

```
1 method howlong(s) {
2   if s.is_a(string) then
3     #{⟦s⟧ ⊆ {string}}
4     s.length()
5   end                    ①
6 }
```

```
1  #{⟦l.@value⟧ ⊆ {string}}
2  method format(l) {
3    l.get(0) + ": " +
       l.get(1).to_string()
4  }
                ③
```

```
1  method do(b) {
2    if b then
3      x := "foo";
4      y := "bar"
5    else                   ②
6      x := 27;
7      y := 15
8    end
9    #{(⟦x⟧ ⊆ {numeric} ∧ ⟦y⟧ ⊆ {numeric})
         ∨(⟦x⟧ ⊆ {string}∧⟦y⟧ ⊆ {string})}
10   x + y
11 }
```

of the if statement due to the assignments and the standard rule for conditionals in Hoare logic states that a postcondition of both branches is also a postcondition of the entire statement.

Mixed-Type Container Elements. In dynamically typed languages, containers like lists are commonly used as records. Since type systems usually enforce all elements to be of the same type, such programs are not typable. In example ③, passing the list `["apples", 5]` to the method `format` would give `l.get(0)` the type `{string, numeric}` as instances of both classes are present in `l`. Consequently, the subsequent `+` operation yields a type error.

Now suppose the method `List.get(n)` had a postcondition `n == 0 → result = self.@value`. We could then conclude {`true`}`l.get(0)`{`result = l.@value`} by substitution and the rule of consequence. Since this implies ⟦`l.get(0)`⟧ = ⟦`result`⟧ = ⟦`l.@value`⟧, adding the precondition in line 1 allows us to conclude ⟦`l.get(0)`⟧ ⊆ `{string}`.

For lists starting with a string the precondition can easily be established and thus our program proven to be free of type errors.

Discussion. The sketched approach eases verification of dynamically typed languages by integrating type inference into the verification environment. We hope that the effort required for verifying dynamically typed programs will in many cases match their statically typed counterparts. For more complex typing problems the algorithm requires the user's help but should still be able to reduce the effort significantly.

References.

[1] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[2] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for javascript. In *POPL*, pages 31–44, 2012.

[3] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *ECOOP*, pages 329–349, 1992.

[4] Matthew J. Parkinson and Gavin M. Bierman. Separation logic for object-oriented programming. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 366–406. Springer, 2013.