Position Paper: Dynamically Inferred Types for Dynamic Languages

Jong-hoon (David) An[1], Avik Chaudhuri[2], Jeffrey S. Foster[3], and Michael Hicks[3]

[1] Epic Systems Corporation, Madison, WI, USA
[2] Advanced Technology Labs, Adobe Systems, San Jose, CA, USA
[3] University of Maryland, College Park, USA

Over the past few years we have been developing Diamondback Ruby (DRuby), a tool that brings static type inference to Ruby [1], a dynamically typed, object-oriented language. Developing DRuby required creating a Ruby front-end, which was extremely challenging: like other dynamic languages, Ruby has a complex, yet poorly documented syntax and semantics, which we had to carefully reverse-engineer. Writing our front-end took well over a year, and now that Ruby 1.9 is available, we are faced with the daunting prospect of significant additional effort to discover how the language has changed and to extend our front-end accordingly. We suspect that maintaining a static analysis system for other dynamic languages, such as Perl or Python, is similarly daunting.

To remedy this situation, we recently introduced a new program analysis technique for dynamic languages: *constraint-based dynamic type inference*, which requires no language front-end, but instead uses introspection features to gather information at run time and infer static types [2]. More precisely, at run-time we introduce type variables for fields, method arguments, and method return values. As values are passed to those positions, we dynamically *wrap* them in proxy objects to track the associated type variables. We also allow methods to have trusted type annotations, which are maintained dynamically at run time. As wrapped values are used, we generate *subtyping constraints* on the associated type variables. We solve those constraints at the end of one or more program runs, which produces a satisfying type assignment, if one exists. Importantly, despite relying on dynamic runs, we can prove a soundness theorem: if the dynamic runs from which types are inferred cover every path in the control-flow graph of every method of a class, then the inferred types for that class's fields and methods are sound for all possible runs. (Currently, this coverage criterion must be checked manually, though we could potentially automate the check.) Note this coverage criterion is in contrast to requiring that every *program* path is covered.

We have implemented this technique for Ruby, as a tool called Rubydust (where "dust" stands for dynamic unraveling of static types). An important property of Rubydust is that, rather than a standalone tool, it is a Ruby library that is loaded at run-time just like any other library. To operate, Rubydust uses Ruby's rich introspection features to wrap objects, intercept method calls, and store and retrieve any type annotations supplied by the programmer. Thus far, we have run Rubydust on a number of small programs, and have found

that Rubydust produces correct, readable types. We expect that our approach could be implemented in the same way in other languages that have sufficient introspection facilities.

We believe that it is worth exploring whether constraint-based dynamic type inference is a practical means to adding static typing support to dynamic languages. In particular, we believe that users will often want to develop their scripts without types at first, and then might like to add types (as checked annotations) later. Constraint-based dynamic type inference could be quite useful for discovering possible annotations automatically. We would hope that Rubydust's approach, made practical, could be applied to other dynamic languages such as Python or Perl, and to gradually typed languages such as ActionScript.

Before we can claim victory, however, there are a number of challenges that require further research. We list a few here. First, Rubydust's performance overhead is significant: an instrumented program can be as much as 1000× slower than the uninstrumented original. We believe the major source of slowdown is in the additional levels of indirection introduced by wrapping all objects. Thus we are currently investigating ways to improve performance by integrating wrapping directly into the Ruby interpreter. Second, Rubydust's inference is currently limited to nominal and structural types involving unions; Rubydust cannot infer intersection or polymorphic types, though it can understand such types in annotations. Our experience with DRuby gives us reason to believe that inferring intersection and polymorphic types would be very useful, but it makes the inference problem significantly harder. Third, inference relies on instrumenting the running program, but some type-relevant events escape instrumentation. In particular, we know of no way to intercept calls to blocks (i.e., closures), nor do we yet know how to *re*instrument a program after it has called `eval` to create new code or definitions. (We suspect this might be accomplished by redefining `eval`, though we have not worked out the details.) Finally, we wish to understand the practical benefits of types: once we have them, what do we do with them? One possibility is to check them, e.g., at method call boundaries. This would permit reporting errors earlier, and the results might be more informative. Another question is whether we have chosen the right design point for our type language: should types be more expressive, to convey richer properties, or perhaps less expressive, so they are easier to read? When are programmers most interested in types, e.g., during maintenance, or during initial development? Many of these questions require careful human studies, which we plan to undertake once we have worked out some technical issues.

References

1. Michael Furr. *Combining Static and Dynamic Typing in Ruby.* PhD thesis, University of Maryland, College Park, 2009.
2. Jong hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic Inference of Static Types for Ruby. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Austin, TX, USA, January 2011.