Documentation Dataflow And Abstract Interpretation, Automatic Program Analysis

Stijn van Drongelen (3117537), Tom Tervoort (), Gerben van Veenendaal (3460692)

June 3, 2013

Introduction

For this assignment, we chose to implement *soft typing* for the programming language Lua. Lua is very popular as a simple scripting language, for example to implement plugins for various programs. We chose Lua because it is a very elegant language, simple, yet powerful.

The implementation of this analysis was done in a combination of Lua and Haskell. The Lua portion is only responsible for parsing Lua code; we borrowed the code from an existing Lua parser by Mark Langen[1].

Lua's type system

Ignoring advanced concepts, Lua has only six types:

- `nil` is used to signify nonexistence. For instance, any non-existing variable has the value `nil`. One peculiarity that illustrates the "nonexistence" semantics is that trying to insert a value with index `nil` into a table leads to an error. We say that `nil` is of type *nil*.
- The *boolean* type is inhabited by `true` and `false`.
- The *number* type is the set of all possible double-precision floating point numbers.
- The *string* type is the set of all sequences of "characters", where one character is at least eight bits.[2]
- The *table* type is a mutable mapping from any non-`nil` value to any non-`nil` value.[3]
- Lua functions are first-class citizens, so we also need to consider them as inhabitants of the *function* type.

Although Lua's semantics are lenient when it comes to its types, some operations can lead to type errors. For example, arithmetic operators only work with numbers, table operations fail with anything besides tables, and only functions are callable. These properties can be used to approximate which types the variables in the program may have if the program won't crash on a type error.

[1] `https://github.com/stravant/LuaMinify`

[2] More information about Lua strings: `http://lua-users.org/wiki/LuaUnicode`

[3] Any key value that is not explicitly in the table, including `nil`, maps to `nil`.

Analysis

In our analysis, we keep track of the possible types a variable may have, with the assumption that the program will not result in a type error. The type lattice we use is product the lattices of the six Lua types, using the product order ($(a, b) \leq (a', b') \leftrightarrow a \leq a' \wedge b \leq b'$). The precision of these lattices varies greatly:

- Because the *nil* and *boolean* types have very few inhabitants, we take the powerset of these as their type lattice, which results in exact precision.
- We initially planned to keep track of certain properties of *number*, but we never implemented it, which leaves us with only bottom (may be any number) and top (may be no number).
- For *string*, we consider small sets of constants to be between top (all possible strings) and bottom (not a string at all). If the set becomes too large after a join, we replace the result with top.

  Implementing this allowed for a simplification in treating tables. Even when only single string constants are allowed in the lattice, we can treat the notation `table.member` as `table["member"]` without any loss of precision.
- The lattice for our *table* type is the most intricate in our system. Besides the bottom (not a table) and top (everything may map to anything), we keep track of the values behind (known) constant indices and (unknown) variable indices.
- Although our *function* type lattice is relatively complex as it is, keeping track of admissable types of parameters and return values, and effects on global variables, we don't actually use it. Effectively, we only use the bottom and top.

Strong points

One strong point of our implementation is that we support a big subset of the language.

Weak points

Running the examples