go oracle: design

http://golang.org/s/oracle-design

Alan Donovan
adonovan@google.com
Created August 25, 2013
Last updated February 13, 2015

This document (and the oracle tool) are obsolete, superseded by guru.
See the new documentation at: http://golang.org/s/using-guru

The **go oracle** is a source analysis tool, typically invoked by an editor, that answers questions about Go programs. This document describes its design. You should read the user manual first.

Motivation

Programmers spend a great deal of of time reading programs. Indeed, most of the time it takes to “write” a program is spent reading it, or more precisely, reading it and making logical deductions about what it does. Logic is, by definition, mechanical, and the goal of this work is to

automate some of the deductions that Go programmers do, day in, day out in the course of their work so that, just as they no longer worry about whitespace and indentation (thanks to gofmt), they should no longer have to worry about a number of other mundane code comprehension tasks that a machine is better suited to carry out: locating definitions, ascertaining types of expressions, deducing the “implements” relation, computing method sets, finding callers/callees, jumping through channels, understanding aliasing.

Traditional IDEs such as Eclipse and IntelliJ support many of these kinds of queries (though not alias analysis---see below) but they are “omnivores”: heavyweight programs imposing a complex theory of project organization on their users, and incorporating editors, build systems, debuggers, source control tools, code browsers and other utilities. We do not wish to pursue this approach for Go because the costs of supporting any specific IDE are very high and yet only the relatively few developers who use that IDE seriously are likely to benefit at all.

Instead, our approach with the oracle is to build a lightweight query system, not coupled to any particular environment, that can be called from almost any editor via a small amount of plumbing. The editor provides only the cursor position or selection and the query (e.g. “what’s the method-set of the selected expression?” or “who sends values on this channel?”) and the result, consisting of both source locations and informational messages, is printed in a compiler diagnostic format that the editor can mark up with hyperlinks in its usual way.

Queries

Here is a partial list of queries we plan to support.

- What is the type of this expression? What are its methods?
- What’s the value of this constant expression?
- Where is the definition of this identifier?
- What are the exported members of this imported package?
- What are the free variables of the selected block of code?
- What interfaces does this type satisfy?
- Which concrete types implement this interface?

And:

- What are the possible concrete types of this interface value?
- What are the possible callees of this dynamic call?
- What are the possible callers of this function?
- What objects might this pointer point to?
- Where are the corresponding sends/receives of this channel receive/send?
- Which statements could update this field/local/global/map/array/etc?
- Which functions might be called indirectly from this one?

In general, queries in the second group cannot be answered exactly because the answer may depend upon dynamic properties of the program; some static approximation is necessary. One approach is to use a **type-based approximation**. For example, we can approximate the set of potential callees at a dynamic function call by assuming, conservatively, that any function of the

appropriate type may be called. For functions with unusual types, this gives good results, but for functions with common types this results in many spurious call edges; a tool that answered the query “what object might this *int pointer point to?” with the response “any int variable whose address is ever taken” would be unlikely to find many users.

Pointer analysis

A more precise approximation can be obtained using **pointer analysis**. This is a static analysis technique in which all the statements of the entire program are analysed for their effect on *aliasing*, the relationship between pointers and the things they point to. There is a huge range of pointer analysis techniques in the literature, but generally, the more precise the technique (i.e. the fewer spurious results it delivers), the more expensive it is to compute.

Fortunately Go presents an attractive target for pointer analysis for several technical reasons:

- Go programs are relatively type-safe.
  Though some do use unsafe.Pointer conversions, these typically appear far less frequently than the analogous cast operations in C or C++.
- Go type hierarchies are shallow.
  Compared to Java, in which very deep concrete class hierarchies are normal, implementation inheritance is impossible in Go, so all concrete types appear only on the lowest level of the type hierarchy.
- Go programs are not dynamically self-extending.
  Go has no dynamic code loading/generating mechanism like `dlopen` or `mprotect(EXEC)` in C or `Class.forName` in Java, so it is feasible to analyse the complete source for the whole program statically.
- Go has no parametric polymorphism.
  Utility functions called from many places with different types present a bottleneck for pointer analysis and demand (more expensive) context-sensitive analysis.

Go programs are also smaller than C++ and Java programs; this may just be a consequence of the relative maturity of the language, but perhaps there is some effect of Go culture here too.

The oracle uses an **inclusion-based** pointer analysis, meaning that when it encounters a statement y = x, the analysis deduces that the set of things to which y may point *includes* (⊆) the set of things to which x may point. An alternative family of analyses is called **unification-based** because they pessimistically conclude that x and y may point to exactly the same set of things, i.e. they are *unified* (≡). Unification-based analyses scale well (linearly in the size of the program) but at the cost of poor precision. In contrast, solving an inclusion-based pointer analysis problem, which requires the computation of the transitive closure of a directed graph of inclusion constraints, requires cubic time and quadratic space in the size of the graph, so for many years it was considered impractical for large programs. However, recent advances in *presolver optimisation* have made it effectively linear by exploiting symmetry and redundancy in the graph; the solver is still $O(n^3)$, but the value of n that it sees is much smaller.

The pointer analysis is mostly **context-insensitive**, meaning that most functions are analyzed exactly once, so the analytic effect of each call to a function combines the effects of all calls to that function; only a few calls (e.g. built-ins, reflection, and some smaller functions) are treated context sensitively. Of course, this reduces precision, but generalized context-sensitive treatment is computationally very expensive.

As a prerequisite to pointer analysis, the program must first be converted from typed syntax trees into a simpler, more explicit **intermediate representation** (IR), as used by a compiler. We use a high-level **static single-assignment** (SSA) form IR in which the elements of all Go programs can be expressed using only about 30 basic instructions.

The pointer analysis is **flow-insensitive**, meaning that the order of the statements (sequencing, loops, conditionals) in the program has no effect. Such analyses are capable of answering *may alias* queries ("may P point to X?") but not *must alias* queries ("must P point to X?"). While true flow-sensitive analysis is very expensive, the use of SSA-form IR gives a degree of flow sensitivity for free. For example, in the sequence

```
p = &x; *p = A
p = &y; *p = B
```

SSA renaming would break p apart into two distinct variables, the first pointing only to x and the second only to y.

Processing a query

The steps required to process a query depend upon the mode of the query and also the selected user input. Typical queries involve the following steps:

- **Converting the filename and byte offset(s)** to an AST node.
  This is done by walking the abstract syntax tree to find the smallest subtree that completely encloses the selection. Whitespace adjoining a subtree is treated as part of the subtree to make the tool more tolerant of sloppy selections.
- **Ascertaining the most appropriate AST node** for the query.
  Certain queries require a particular type of node, such as a function call or a channel send or receive. The tool walks up (or sometimes down) the AST, starting at the selected node, to determine the appropriate node.

Many queries need only typed ASTs: those that connect definitions with references, enumerate the members of a package or method-set, or show the value of a constant expression. Such queries can be completely processed at this point. For queries needing pointer analysis, the following additional steps are required:

- **Locating the SSA value** corresponding to the source expression.
  The SSA builder does some bookkeeping to record the relationship between syntax trees and SSA values (globals, parameters and instructions), analogous to a compiler maintaining debug information.
  There may be zero, one or many SSA values for a given expression: zero if the expression is trivially dead code, one in the common case, or many if the function is

analyzed multiple times due to context-sensitivity.
Each occurrence of a variable identifier is treated distinctly to obtain the local flow-sensitivity mentioned above.

- Formulating and solving a **pointer analysis** problem.
  The pointer analysis query includes the set of SSA values of interest: for a "describe" query this is just the value of the selected expression; for a channel send/receive query, it would be all values `ch` for which there is a `<-ch` or `ch<-x` instruction anywhere in the program.
- **Displaying the result.**
  Some bookkeeping is required to transform the results of pointer analysis into the desired output, such as a call graph, or a set of "may alias" facts about channel send and receive statements.

Analysis libraries

The oracle uses the following libraries in the golang.org repo:

`go/{token,scanner,ast,parser}`	parser
`golang.org/x/tools/go/types`	type checker
`golang.org/x/tools/go/loader`	source package loader
`golang.org/x/tools/go/ssa`	SSA IR
`golang.org/x/tools/go/pointer`	pointer analysis
`golang.org/x/tools/oracle`	oracle library
`golang.org/x/tools/cmd/oracle`	oracle command

The SSA and pointer analysis libraries were developed for and with the oracle, but have been used for other applications, like improved documentation tools (e.g. godoc -analysis=type,pointer) and the front-end of the llgo compiler.

Restrictions

Well-formed inputs are required for pointer analysis

Since the SSA builder requires a well-formed Go program as its input, the oracle cannot tolerate type errors when doing pointer analysis queries.

Complete Go sources are required for pointer analysis

Pointer analysis requires that the complete Go source of the program is available. The effects of functions written in assembly or C cannot be observed by the analysis, causing imprecise or unsound results, although often these problems only appear in the vicinity of the native code.

go/build package layout conventions are assumed

The oracle uses the `go/build` package to find the source files for a given package. However, the 'go test' command has significant additional logic for constructing test packages and the

oracle cannot faithfully reproduce this in all cases (yet). Furthermore, some proprietary build systems (such as Google's) have a different package organization requiring a different algorithm to locate the source files given an import path. The oracle cannot be run in such environments (yet).

Status

Future work

Usability improvements

- **describe** on a named return parameter gives an error (can't find SSA value).
  It should be equivalent to selecting either (a) an explicitly declared local variable or (b) the union of corresponding return operands.
- Include complete [start...end) extent information in JSON/XML output, not just a position, where available. (One subtlety is that sometimes a position is better than an extent. Consider the expression e1 + e2 where e1 and e2 are complex expressions: if you want to indicate the addition operator, the position of the "+" token is much more helpful than the entire extent of the expression.)
- A type-based call graph could provide (less precise) answers to callers/callees queries without a scope.
- Allow the tool to be run on edited but unsaved code, substituting an editor buffer for a file on disk.

Analysis features

- Add an **updates** query: "which statements can update this (addressable) expression?"
- Add a **creators** query: "which statements create an object of this type?"

Performance optimisations

- Ideally the tool should respond almost instantaneously to most queries, and we believe this is quite feasible for all but the largest programs, without resorting to stateful or long-lived analysis processes.