go oracle: user manual

Alan Donovan
adonovan@google.com
August 25, 2013

The **go oracle** is a prototype source analysis tool that answers questions about Go programs. This document explains how to use it.

There is also a design document.

The oracle may be invoked directly from the command line, or indirectly via an editor that provides the tool with the current cursor position/selection plus the kind of query you wish to perform.

How it helps

The oracle is designed to fully automate the answering many of the questions about elements of your program that come up all the time during a typical day of programming. Questions such as:

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 will a value sent on this channel be received?
Which statements could update this field/local/global/map/array/etc?
Which functions might be called indirectly from this one?

In many cases, using the oracle is as simple as selecting a region of source code, pressing a button, and receiving a precise answer to the query almost immediately.

The first set of questions above can be answered using only local, modular reasoning by looking at the syntax tree, the types, or the members of a single package, but queries in the second set depend, in general, upon **global properties** of your application requiring more analytical effort---human or robot---to deduce.

Caveats

The oracle is a **prototype**. Its user interface has many rough edges and will almost certainly need major changes. The analysis libraries may contain bugs. Configuration is trickier and less flexible than it ought to be. Analysis is an order of magnitude slower than our goal. Nonetheless, the oracle offers some analytical services that advance the state of the art in code comprehension tools---for any language---and adventurous users may find it useful today. Please report bugs directly to the author for now.

Building it

Run the following command to build the oracle:

```
 % go get code.google.com/p/go.tools/cmd/oracle
```

This will cause an executable named oracle to appear in your $GOPATH/bin directory.

Command syntax

Running the oracle with no arguments prints a summary of the command syntax.
Here's an example command invoking the oracle:

```
  % oracle -mode=describe -pos=src/pkg/net/http/triv.go:#1042,#1050 -format=json \
          src/pkg/net/http/triv.go
```

There are four inputs of interest:

- The **mode** of the query, `-mode=describe` in this example.
  Each supported query is described in its own section below.
- The **position** of the cursor or selected syntax, as a flag of the form `-pos=file:#start,#end` where [*start*, *end*) forms a half-open interval of byte indices within *file*, starting at zero.
- The desired **output** format. Supported formats include:
  - `plain`, a human-readable format resembling typical compiler diagnostic output
  - `json`, a structured data format specified at go.tools/oracle/json/json.go
- The **scope** of the analysis. In the simplest case, this is just the 'main' package of your program. In this example, it's a package containing only the single file `src/pkg/net/http/triv.go`. This concept is expanded in the next section.

Analysis scope

For global queries (explained above), the oracle needs to know which package defines your application's main function, and it needs source code for all packages that are transitively imported from it---it cannot analyse isolated libraries. We call this the **scope** of the analysis.

The scope may consist of several programs, such as a client and a server, or all the programs you routinely work on, or a set of libraries and their tests. Bigger scopes are better because they cause the analysis to visit more code. If a library function is not reachable in a given scope, the analysis can't answer any questions about it---as if it's not there. This is analogous to the way a linker works: library functions that cannot be called in a given executable are discarded.

Example scopes:

1. `fmt`
   the import path of a library defining one or more Test* functions
2. `code.google.com/p/go.tools/cmd/oracle`
   the import path of a package with a main entry point.
3. `src/pkg/net/http/triv.go`
   an ad-hoc main package consisting of a single file.
   (Multiple files in the same package should be separated with commas.)

All three of these may be specified in the same command. However, due to a current limitation of the type checker, only the first of the import-path style arguments will contribute any tests to the pointer analysis scope.

Warnings

If the static analysis must make an assumption that it cannot prove (e.g. about the behaviour of native code that it cannot see or unsafe.Pointer conversions it cannot understand) it will print a warning after its results. Currently the standard libraries cause the oracle to print many warnings, so the output can be rather noisy. These will diminish over time as gaps in the analysis are filled.

Editor integration

The oracle may be invoked from any editor capable of running an external tool (such as a compiler) and displaying its output. Since many editors treat file names appearing in compiler diagnostics as hyperlinks to the location of the error, the oracle prints its answer using a similar syntax when invoked with `-format=plain`.

Currently the only editor for which bindings exist is **Emacs**, though we hope to add support for others based on demand. Please contact the author if you'd like to help connect the oracle to another editor such as Vim, Acme or Eclipse.

Emacs

Emacs expects to find the oracle executable in $GOROOT/bin, not where `go get` places it (i.e. P/bin where P is the first directory named by $GOPATH), so build it using the `go get` command above and then move it:

```
      % mv $GOPATH/bin/oracle $GOROOT/bin/
```

(This command assumes your GOPATH consists of exactly one directory; adjust accordingly if yours has several.)

Within Emacs, load the oracle.el file using a command such as this:

```
   M-x load-file $GOPATH/src/code.google.com/p/go.tools/cmd/oracle/oracle.el
```

Typically, users will add this command to their ~/.emacs startup configuration.

Before you can run the oracle, you must tell Emacs the analysis scope, which is done using the command:

```
  M-x go-oracle-set-scope
```

This command prompts you for the analysis scope, described above, with words separated by spaces. The effect of `go-oracle-set-scope` persists across all oracle invocations until it is called again with a different value.

To invoke the oracle, position the cursor on (or select) the syntax of interest and call the Emacs command go-oracle-*xxx* where *xxx* is the mode of the query. For example:

```
  M-x go-oracle-callees
```

The most commonly used query, **describe**, is bound to the shortcut key `<F4>` . To test your configuration, load a file within your analysis scope, select an expression, and hit `<F4>`. After a moment a window should appear with the results, looking something like this:

```
Go Oracle
▶       reference to var result string
▶       defined here
```

Queries

This section describes the set of oracle queries. See the Table of Contents for the complete list.

Notation for examples

In the examples, source code is shown in grey, user-selected source code is highlighted in yellow, and `-format=plain` tool output is colored blue. In the actual tool output, each line is preceded by the source location most relevant to it, but to avoid distracting detail in the examples, the file names have been rendered as a ▸ symbol. In some cases, location markers such as L1 have been added to make the source/results correspondence clear.

For brevity, the `-format=json` output is not shown, but it contains essentially all the same information as the `plain` output, broken down into a tree of structured data for ease of parsing. Read the `go.tools/oracle/json` documentation for more details.

---

callees

The **callees** query shows the possible call targets of the selected function call site. The cursor or selection must be within a function call expression.

Example: a callees query on the main dispatcher of `net/http`'s trivial webserver reveals all the handlers that are registered by the application.

```
    func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
            f(w, r)
    }
```

```
▸       this dynamic function call dispatches to:
▸               net/http.NotFound
▸               main.FlagServer
▸               main.ArgServer
▸               expvar.expvarHandler
▸               main.DateServer
▸               main.HelloServer
▸               main.Logger
```

- func@1247.21

(The last one is an anonymous function.)

Example: a callees query on an interface method call (itself in an anonymous callback) reveals the sole target of the call.

```
func StripPrefix(prefix string, h Handler) Handler {
	if prefix == "" {
		return h
	}
	return HandlerFunc(func(w ResponseWriter, r *Request) {
		if p := strings.TrimPrefix(r.URL.Path, prefix); len(p) < len(r.URL.Path) {
			r.URL.Path = p
			h.ServeHTTP(w, r)
		} else {
			NotFound(w, r)
		}
	})
}
```

- this dynamic method call dispatches to:
- (*http.fileHandler).ServeHTTP

---

callers

The **callers** query shows the possible callers of the function containing the selection.

Example: FlagServer is an HTTP handler function in the trivial webserver in `net/http`. A callers query on that function reveals where the webserver dispatches requests to it.

```
func FlagServer(w http.ResponseWriter, req *http.Request) {
        w.Header().Set("Content-Type", "text/plain; charset=utf-8")
        …

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
        f(w, r) // L1
        ...
```

```
▸       main.FlagServer is called from these 1 sites:
▸ L1            dynamic function call from (http.HandlerFunc).ServeHTTP
```

Example: a second callers query, this time on ServeHTTP:

```
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
        f(w, r)
        ...

func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request) {
        ...
        h, _ := mux.Handler(r)
        h.ServeHTTP(w, r) // L2
}
```

```
▸       (http.HandlerFunc).ServeHTTP is called from these 1 sites:
▸ L2            dynamic method call from (*http.ServeMux).ServeHTTP
```

---

callgraph

The **callgraph** query shows the call graph of the entire program, which may have many thousands of nodes and edges. The selection is ignored.

Example: here is an excerpt of the call graph of a program that uses the `fmt` package.

```
            …
▸     278       log.Fatalf
▸     279           fmt.Sprintf
▸     280               fmt.newPrinter
▸     281                   (*fmt.cache).get
▸                               (*sync.Mutex).Lock (52)
▸                               (*sync.Mutex).Unlock (56)
▸     282                       func@161.23
▸     283                   (*fmt.fmt).init
▸     284                       (*fmt.fmt).clearflags
▸     285               (*fmt.pp).doPrintf
                        …
```

The graph is rendered as a spanning tree, using indentation to show parent/child relationships. When a node appears for the first time, it is prefaced by a fresh number (e.g. 278 for log.Fatalf) and when it appears again, the number is shown after (e.g. 52 for (*sync.Mutex).Lock).

A function may have multiple nodes in the call graph if the analysis decides to treat it *context sensitively*: in effect it makes two distinct copies of the function based on where it is called from. This can often improve the precision of the analysis. Look at the numbers if in doubt.

callstack

The **callstack** query shows an arbitrary path from the root of the callgraph to the function containing the selection. This may be useful to understand how the function is reached in a given program.

Example: the result of a callstack query from the ServeHTTP function in `net/http`.

```
func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
        ctr.mu.Lock()
        ...
}
```

```
▸    Found a call path from root to (*main.Counter).ServeHTTP
▸            (*main.Counter).ServeHTTP
▸            dynamic method call from (*http.ServeMux).ServeHTTP
▸            dynamic method call from (http.serverHandler).ServeHTTP
▸            static method call from (*http.conn).serve
▸            static method call from (*http.Server).Serve
▸            static method call from (*http.Server).ListenAndServe
▸            static method call from net/http.ListenAndServe
▸            static function call from main.main
```

The precision and usefulness of callstack information varies considerably, especially if the call path contains a greater degree of dynamic calls. The chosen callstack might be infeasible, i.e. never occurring during any execution.

describe

The **describe** query shows various properties of the selected syntax: its syntactic kind, type, method set, constant value, point of definition, points-to set, etc, as appropriate. Almost any piece of syntax may be described, and the oracle will try to print all the useful information it can.

Example: a describe query on a field selection expression in the `net/http` package.

```
        func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
                ...
                io.Copy(buf, req.Body)
                …
    }
```

```
▸   reference to var Body io.ReadCloser
▸   defined here
▸   interface may contain these concrete types:
        *struct{*strings.Reader; io.Closer}, may point to:
▸           complit
▸       *http.body, may point to:
▸           complit
▸           complit
▸           complit
▸       *http.expectContinueReader, may point to:
▸           complit
```

The response to this query contains the type of the expression, the location of the definition of the struct field, the list of concrete types that it (an interface) may contain, and for each of those concrete types, all of which are pointers, the set of objects to which it may point, with source locations where available. (`complit` indicates the object allocated by a composite literal.)

Example: from package `net`.

```
func (h *dnsHeader) Walk(f func(v interface{}, name, tag string) bool) bool {
    return f(&h.Id, "Id", "") && …

func (dns *dnsMsg) Pack() (msg []byte, ok bool) {
    var dh dnsHeader // L1

func (dns *dnsMsg) Unpack(msg []byte) bool {
    var dh dnsHeader // L2
```

```
▸       unary & operation of type *uint16
▸       value may point to these labels:
▸ L1            dh.Id
▸ L2            dh.Id
```

The syntax “dh.Id” denotes the .Id field of the object created by the identifier dh. In fact there are two distinct local variables, both called dh, into which this expression may point.

Example: an excerpt of a describe query on a package name. Essentially the same results are obtained when the selection is the import path or when it is the package identifier.

```
        import "net/url"
        var u url.URL
```

```
▸       import of package "net/url"
▸               type  Error                    struct{...}
▸                       method (*url.Error) Error() string
▸               type  EscapeError              string
▸                       method (url.EscapeError) Error() string
▸               func  Parse                    func(rawurl string) (url *url.URL, err error)
▸               func  ParseQuery               func(query string) (m url.Values, err error)
▸               func  ParseRequestURI          func(rawurl string) (url *url.URL, err error)
▸               func  QueryEscape              func(s string) string
▸               func  QueryUnescape            func(s string) (string, error)
▸               type  URL                      struct{...}
▸                       method (*url.URL) IsAbs() bool
▸                       method (*url.URL) Parse(ref string) (*url.URL, error)
▸                       method (*url.URL) Query() url.Values
                        ...
```

The description of a package includes all its exported members, their types, methods, and values (for constants). If the current package is described (by selecting the package declaration package p), the description includes the non-exported members too.

freevars

The **freevars** query enumerates the free variables of the selection. "Free variables" is a technical term meaning the set of variables that are referenced but not defined within the selection, or loosely speaking, its inputs.

This information is useful if you're considering whether to refactor the selection into a function of its own, as the free variables would be the necessary parameters of that function. It's also useful when you want to understand what the inputs are to a complex block of code even if you don't plan to change it.

To make the results more useful, the output of the query differs slightly from the textbook definition of free variables:

- the output does not report any names defined at package level, since they would not need to be passed as parameters to a function;
- for each free struct variable, the output reports each distinct access path (e.g. s.x.y) as a free variable.
- the output also reports references to free constants and types.

Example: the free variables of the body of a loop in the `strings` package.

```
// Second pass: find repeats of pattern's suffix starting from the front.
for i := 0; i < last; i++ {
        lenSuffix := longestCommonSuffix(pattern, pattern[1:i+1])
        if pattern[i-lenSuffix] != pattern[last-lenSuffix] {
                // (last-i) is the shift, and lenSuffix is len(suffix).
                f.goodSuffixSkip[last-lenSuffix] = lenSuffix + last - i
        }
}
```

```
▸ Free identifiers:
▸ var i int
▸ var last int
▸ var pattern string
▸ var f.goodSuffixSkip []int
```

implements

The **implements** query shows the *implements* relation for all interfaces and concrete types defined in this package. The selection is ignored.

Example: an excerpt of the *implements* relation of the `io` package.

- Interface io.Writer:
    - *io.PipeWriter
    - *io.multiWriter
- Interface io.Seeker:
    - *io.SectionReader
- Interface io.Closer:
    - *io.PipeReader
    - *io.PipeWriter

peers

The **peers** query shows the set of possible sends/receives on the channel operand of the selected send or receive operation; the selection must be a `<-` token.

Example: a peers query on a receive operation in the `net/http` package.

```
var textprotoReaderCache = make(chan *textproto.Reader, 4)     // L1

func newTextprotoReader(br *bufio.Reader) *textproto.Reader {
        select {
        case r := <-textprotoReaderCache:  // L3
                r.R = br
                return r
        default:
                return textproto.NewReader(br)
        }
}

func putTextprotoReader(r *textproto.Reader) {
        r.R = nil
        select {
        case textprotoReaderCache <- r:     // L2
        default:
        }
}
```

```
▸       This channel of type chan *textproto.Reader may be:
▸ L1            allocated here
▸ L2            sent to, here
▸ L3            received from, here
```

referrers

The referrers query shows the set of identifiers that refer to the same object as does the selected identifier, within any package in the analysis scope.

Example: find all references to a function parameter in the `fmt` package.

```
func (p *pp) fmtUint64(v uint64, verb rune, goSyntax bool) {
        switch verb {
        case 'b':
                p.fmt.integer(int64(v), 2, unsigned, ldigits)  // L1
        case 'c':
                p.fmtC(int64(v))  // L2
        case 'd':
                p.fmt.integer(int64(v), 10, unsigned, ldigits) // L3
        …
```

```
▸       defined here as var v uint64
▸ L1    referenced here
▸ L2    referenced here
▸ L3    referenced here
        ...
```

Troubleshooting

- ***The oracle says that function F is reachable, but I know that it's not.***
  The oracle says that pointer P can point to label L, but I know that it cannot.

This class of errors arise from "false positives" or imprecision in the pointer analysis. A **sound** pointer analysis may make conservative approximations when it isn't capable of fully capturing the behaviour of your program. These kinds of false reports are mostly not considered bugs, although of course if they are too numerous, the usefulness of the tool may be diminished.

- ***The oracle says that function F is dead code, but I know that it's not.***
  The oracle says that pointer P may not point to label L, but I know that it can.

This class of errors arise from "false negatives" or **unsoundness** of the pointer analysis, and they generally indicate a bug. Reflection is not currently supported, leading to unsound results (missing edges in the call graph and underestimates of points-to sets); this will be fixed in due course. unsafe.Pointer conversions are also not supported, and may never be.

Don't forget that the pointer analysis only looks at code reachable in the analysis scope that you specified, e.g. the entire program whose 'main' package was named on the command line. Even a large Go program might use only small parts of some of the libraries it depends upon, so pointer analysis queries about the unused parts will return null results. This is correct, and the expected behaviour. Specifying a larger scope (more main packages and tests) can improve the analytical coverage of your libraries.