GOAL: A SYSTEM FOR GO PROGRAM ANALYSIS USING RELATIONAL DATABASES

By

Adam Domurad

Faculty of Science, University of Ontario Institute of Technology

April 11th, 2014

An undergraduate thesis/directed studies submitted to the
University of Ontario Institute of Technology
in accordance with the requirements of the degree
of Bachelor of Science (Hons) in the Faculty of Science

Abstract

Abstract

The Go programming language is a modern systems programming language developed primarily by Google. Since it's inception in 2007, the language has gained popularity as a simple, familiar, performance-oriented language. To help foster growth of complementary tools, the Go development team provides robust components for parsing and analysis Go, written in Go itself.

An experimental Domain-Specific Language (DSL) and associated runtime system, referred to as the **GoAL system** (for **Go Analysis Language**), were designed and implemented [1] for the purpose of aggregating Go program semantic data from a variety of sources. The system provides an event-based mini-language, embedded in Lua, for XPATH-like selection of information about a program element. From there, the selected information can be stored persistently in a relational database and to be queried for performing analysis tasks.

The GoAL system supports connecting to and querying PostGres SQL databases, as well as creating, loading, and querying SQLite3 databases. The SQLite3 driver used includes recent support for recursive Common Table Expression (CTE) definitions.

To demonstrate the usage of this runtime, interface satisfaction for Go was achieved using the GoAL system, and a complementary tool, interfacing with Git, was created.

Contents

List of Figures

Chapter 1

Background

1.1 Go

The **Go programming language** is a modern systems programming language which started development at Google in 2007. The Go language is designed based on principles of minimalism, and has many parallels to a modern spritual successor to C. Despite its relatively recent appearance, it has already proven to be a stable tool for use in production environments. Since inception, development has been rapid with Go version 1.0 being released on 28, March 2012. The whole-version milestone marked Go reaching a level of stability required to allow for forwards-compatible software to be written.

1.2 Lua

The **Lua programming language**, a chosen auxiliary scripting language, is a small, easy-to-embed language, with a light-weight runtime. Lua is often used for configuration and scriptability in a large number of projects. In constrast

to other popular scripting languages such as Python, Lua takes a minimalist approach in standard library design.

By design, Lua provides very little tools in its plain distribution. As a language designed for light-weight embedding, it aims to not provide redundancy with the tools of the system in which it is embedded.

1.3 SQL

SQL is a domain-specific programming language (DSL) designed for managing data held in a relational database management system (RDBMS). Originally based upon relational algebra and tuple relational calculus, SQL consists of a data definition language and a data manipulation language. The majority of popular relational databases to-date employ SQL as their query language. [9]

1.4 Git

Used to create the "GoAL git tool", discussed first in the Design section, Git is a source code management system originally developed alongside the Linux kernel. It features good performance characteristics, strong support for non-linear development, and support for many different varieties of workflows. Usage of the tool is growing steadily. The Eclipse Foundation reported in its annual community survey that as of May 2013, more than 36% of professional software developers use Git as their primary source control system, compared with 27.6% in 2012 and 12.8% in 2011. [10].

1.5 Abstract Syntax Trees

An Abstract Syntax Tree (AST) encodes a parsed representation of a body of source-code, typically associated with the smallest compilation unit considered. For most languages, including Go, this compilation unit is a single file.

This standard representation is used almost universally for compilation and general language analysis due to its ability to represent arbitrarily nested expressions conveniently.

An AST can be said to consist of block-nodes, which contain a variable sized list of children nodes, and label-nodes, which can hold data as well as fixed amount (possibly 0) of children nodes. For example, a procedure declaration in a programming language can be represented as a label-node, pointing to block-nodes holding its parameters & instructions, and containing data representing the function's name.

Chapter 2

Introduction

2.1 The GoAL System

The GoAL system was designed for aggregating data from the provided Go analysis packages, and accessing them all uniformly, filtering information required for analysis. The GoAL system consists of a runtime written in Go, a domain-specific-language (DSL) for using the runtime from Lua, and integration with relational databases such as PostGres or SQLite.

2.2 Using the GoAL System

Analysis of Go programs can be achieved using relational queries alone, by using Lua scripts supported by relational queries, or by using Lua scripts alone. Effective usage of the GoAL system, for a given Go analysis problem, consists of three steps:

1.) Express your target Go analysis problem as an algorithm operating on some complex data formed from simple, non-nested sets of data. These non-

nested sets of data should be expressible as information readily available from a given node. Data can be aggregated from a node and its children.

2.) Gather simple, non-nested data using a GoAL procedure. GoAL procedures are essentially event hooks that activate when a given node is encountered in the Go AST.

The available children of a node can be deduced from the go/ast documentation, located at http://golang.org/pkg/go/ast/.

GoAL procedure select and process data to be output for further processing; they are not intended to be used for the entire analysis task themselves. Since GoAL expressions operate on a node's locally accessible information, they are ideal for tree serialization to relational databases.

Note: If the expressions in GoAL do not suffice in expressing some analysis task, the analysis can be augmented with Lua routines, which provide access to arbitrary Lua state. The Go objects representing AST nodes can be used with almost identical syntax in Lua as they would be used in Go, with regards to member access and method calling. Note only one caveat, Go slices begin 1-indexed in Lua.

3.) Using data aggregated by the GoAL procedure, query the data using a combination of SQL and procedural Lua. The non-nested data sets can be joined to form structured data as required.

Thus, data is extracted from a set of Go packages using GoAL procedures, which are written in a special form of Lua (ie, an embedded DSL). These procedures extract information from the tree, providing convenient operations to store them persistently.

2.3 Rationale

The analysis scheme described is somewhat more complicated than directly operating on the Go AST using Go itself, and forming analysis routines directly. However, there are several benefits to be gained from using the GoAL system along with a relational database:

1.) Inheriting benefits of relational databases

Owing to the usage of a relational database as an intermediate step, the benefits of holding data in a database are inherited.

Several advantages can be identified:

- The data persists, allowing for reuse between program executions without reparsing. SQLite supports in-memory databases as well.

- The relational engine will handle complex features such as query optimization for you (for the most part). Subsets of this data-set can be queried abstractly, with the relational database handling memory management and organization concerns.

- Relational engines are built to scale for very large data-sets, allowing analysis routines to scale to millions of lines of code with a fixed memory footprint.

2.) More convenient data aggregation

Compared to operating directly on the AST, GoAL provides a number of conveniences. Namely, GoAL associates type inference results with each node, and converts Go locations & inferred types to string representations for more convenient serialization.

3.) Creating flexible analysis tools

The GoAL system is potentially advantageous as the basis for creating analysis

tools which themselves expose scriptability and queries, as it provides Lua scripting along with SQL integration. This usage is explored with the "GoAL Git Tool".

Chapter 3

Design

3.1 Motivation: Relational Databases as an Analysis Tool

Relational databases are subject to intense optimization and scrutiny. SQLite has a test suite of 91 million source-lines-of-code (SLOC), dwarfing its 84.3K SLOC implementation. [3].

State of the art techniques for analyzing large bodies of code in languages such as C++ already rely heavily on database technology, such as Eclipse's C++ indexer. [7] Relational databases feature persistence, maturity, and flexibility of retrieval, all which help when developing large-scale static analysis algorithms. In addition to this, relational databases provide a common platform for extensibility, allowing new analysis routines to be written in SQL by anyone with knowledge of the underlying schema.

In this light, a system was designed for aiding analysis of the Go programming language, centered around the use of relational databases.

3.2 Architecture of the GoAL System

The GoAL system's primary purpose was chosen to be the flexible collection of data from Go analysis packages (as implemented, go/parse & go/types) for convenient analysis in a relational database.

Go's AST, encoding a parsed representation of the source-code under analysis, was chosen as the language representation on which to operate. This representation was chosen because it preserves much of the directly observable features of the actual source-code, while providing a data representation that reflects the language's interpretation. Importantly, AST's can be effectively annotated with semantic information about the source-code elements which they represent.

The Go language's standard parsing package (go/parse), along with its AST handling library (go/ast) were chosen to provide in-order traversal over each node of the Go AST. The primary integration with databases was a combination of intelligence about the schema of the database for which data is being serialized, and syntactic convenience for inserting data into these nodes.

To provide maximum flexibility in selecting data from the AST, a dynamic selection scheme was designed using the Lua programming language.

The GoAL system consists of a runtime component ('the GoAL runtime'), as well as a supporting domain-specific language ('the GoAL DSL'). For maximum expressiveness, a domain-specific language ('the GoAL DSL') was designed for selecting information from a node in the parsed AST. The language, while containing its own bytecode and argument stack, looks largely like Lua, and it's syntactic elements can be manipulated as first-class Lua objects. The GoAL DSL was chosen to be bootstrapped by the Lua runtime, providing

parsing logic, and allowing for execution of commands in a script form.

3.2.1 Defining Events in the GoAL System

In GoAL, information is extracted using an event-based system that triggers for operations on each node. The principle method for extracting information from the Go language using the GoAL system is, using a Lua script, defining a series of event hooks containing GoAL procedures.

The Lua 'Event' procedure (provided by GoAL) acts as a root node for denoting GoAL procedures. Calling 'Event' registers an event hook for a specified node type.

```
Event(<ast.Node implementing type from go/ast package> "node") (
    ... operations ...
)
```

The 'Event' Lua routine is the top-level function call for all GoAL event registrations. Without it surrounding a GoAL expression, or some macro that calls it, **GoAL expressions will have no effect**. Additionally, only valid GoAL expressions are allowed inside it. The underlying implementation converts the expression to a custom bytecode format that is mapped to a runtime type from the go/ast Go package.

3.2.2 Ordinary Operators on a Node

The GoAL runtime generally provides access to a node object's data paralleling the structures defined in the go/ast package. Nodes handles are referenced by literal strings. The equivalent of a Go expression A.B.C is B.C "A".

3.2.3 Special Operators on a Node

1. name

Example Usage:

```
Event(FuncDecl "node") (
    Printf("Name was '%s'", name "node")
)
```

Example Result:

```
Name was 'MyFunctionName'
```

For any node that has an associated name, the 'name "node"' operator will resolve to a string holding the node's name. Generally, an equivalent operation (in GoAL) is 'Name.Name "node"'.

2. typeof & basetype

Example Usage:

```
Event(FuncDecl "node") (
    Printf("Type was '%s'", typeof "node")
)
```

Example Result:

```
Type was 'func(mypackage.MyType)'
```

Rather than providing the full depth of access (and resulting complexity) of the go/types library, semantic type information is provided by a simple 'typeof' operation, returning a unique, package-qualified string representation of the node. This is usable in queries very naturally, as type-checking operations can be done with simple equality.

To ease logic that requires comparing types regardless of referenceness, such as interface satisfaction, the 'basetype' operator is provided.

3. location & end_location

Example Usage:

```
Event(BinaryExpr "node") (
    Printf("Defined from '%s' to '%s'", location "node", end_location "node")
)
```

Example Result:

```
Defined from 'foo.go:1:1' to 'foo.go:1:10'
```

The 'location' and 'end_location' operations provide unique source-code locations with positions as well as line numbers. This provides a slight convenience over the raw go/ast library, which requires use of the go/token library to resolve to their line and position.

4. id

Example Usage:

```
Event(FuncDecl "node") (
    Printf("ID was '%d', ID of child block node was '%d'",
        id "node", BlockNode.id "node"
    )
)
```

Example Result:

```
ID was '2', ID of child block node was '3'
```

Returns an ID, unique within this pass. Can be used to dump a reconstructable tree, as the ID's of children nodes can be. It returns 'nil' if passed 'nil'.

3.3 Case

Using Lua, the GoAL system is informed about the schema using the DataSet and Data Lua procedures (provided by GoAL). Within a GoAL expression, the "Store" GoAL operation places data into tables declared with the DataSet procedure.

3.4 Case Study: Interface Satisfaction

An interface satisfaction in Go is defined as a relationship between an interface and a type, whereby a type has all the associated procedures (methods) required by an interface.

An interface satisfaction query & associated GoAL program, was designed for purposes of GoAL system evaluation. The program emits Go interface functions, methods, type declarations, interface declarations, and interface requirements. As well, data about inheritance relationships is encoded.

Complicating the issue, types can be embedded in Go, similar to inheritance in other languages.

A complete example is given for embedding all the information required for performing interface satisfaction:

interface-satisfaction.lua:

```
DataSet("sqlite3",
  "interface-satisfaction.db",
  --[[Delete existing database?]] false
)

-- SQL schema
```

```
Data "methods" (
  Key "name", Key "type", Key "receiver_type", "location"
)
Data "functions" (
  Key "name", Key "type", "location"
)
Data "structs" (
  Key "type", "location"
)
Data "interfaces" (
  Key "interface", "location"
)
Data "interface_reqs" (
  Key "interface", Key "name", Key "type", "location"
)
Data "interface_inherits" (
  Key "type", Key "embedded_type"
)
Data "struct_inherits" (
  Key "type", Key "embedded_type"
)

EventCase(FuncDecl "f") (receiver "f")(
    -- We have a receiver, therefore we have a method
    Store "methods" (name "f", typeof "f", receiver.basetype "f", location "f")
) (Otherwise) (
    -- We have no receiver, therefore we have a function
    Store "functions" (name "f", typeof "f", location "f")
)

-- Syntactic convenience through Lua-based macros:
local function CaseEmpty(var) return Case(Equal(Len(var), 0)) end
```

```
EventCaseType (
    -- Create TypeSpec event, switch on 'Type "n"'
    TypeSpec "n", Type "n"
) (InterfaceType) (
    Store "interfaces" (name "n", location "n"),
    ForAll "f" (Type.Methods.List "n") (
       CaseEmpty(Names "f") (
            -- We inherit requirements of all embedded types
             Store "interface_inherits" (name "n", basetype "f")
       ) (Otherwise) (
             Store "interface_reqs" (name "n", name "f", typeof "f", location "f")
       )
    )
) (StructType) (
    Store "structs" (name "n", location "n"),
    ForAll "f" (Type.Fields.List "n") (
       CaseEmpty(Names "f") (
          -- We inherit methods of all embedded types
           Store "struct_inherits" (name "n", basetype "f")
       )
    )
)
```

After calling the "Analyze" Lua procedure on a given file set (provided by GoAL), the data can be queried. The following SQLite query, compatible with (at least) SQLite 3.8.4.2 and above, successfully solves interface satisfaction.

interface-satisfaction.sql:

```
WITH RECURSIVE
-- Interface embedding:
        iface_embed_closure(iface, embed)
AS (
```

```
        select I.interface, I.interface from interfaces I
UNION
        select C.iface, H.embedded_type
        from iface_embed_closure C join interface_inherits H
        where H.type = C.embed
),
-- Struct embedding:
        struct_embed_closure(type, embed)
AS (
        select I.type, I.type
        from structs I
UNION
        select C.type, H.embedded_type
        from struct_embed_closure C join struct_inherits H
        where H.type == C.embed
),
-- Finding all the requirements of an interface:
        iface_reqs_closure(iface, name, signature)
AS (
        select C.iface, R.name, R.type
        from iface_embed_closure C
        join interface_reqs R
        where C.embed == R.interface
),
-- Finding all the methods of a struct:
        struct_methods_closure(struct, name, signature)
AS (
        select C.type, M.name, M.type
        from struct_embed_closure C
        join methods M
        on C.embed  == M.receiver_type
),
```

```
-- Finding all the satisfactions counts:
        struct_iface_sat_counts(struct, iface, satisfied)
AS (
        select C.struct, R.iface, count(*)
        from struct_methods_closure C join iface_reqs_closure R
        on C.name == R.name and C.signature == R.signature
        group by R.iface, C.struct
),
-- Count the amount of interface requirements:
        iface_req_counts(iface, needed)
AS (
        select C.iface, count(*)
        from iface_reqs_closure C
        group by C.iface
),
-- Find all the satisfactions:
        struct_iface_sat(struct, iface)
AS (
        select C.struct, I.iface
        from struct_iface_sat_counts C join iface_req_counts I
        on C.iface == I.iface and C.satisfied == I.needed
)
-- Selection:
SELECT * from struct_iface_sat;
```

The query relies on finding all the methods, accounting for embedding, and all the interface requirements, accounting for embedding. The GoAL program preceding the query stores struct embedding relationships, and method information by first dropping the detail of whether the type is a reference type or not, relaxing the necessity for handling this detail in the query. To find all the interface satisfactions, the number of methods that a struct has that matches a given type signature are compared to the amount of requirements that that

interface has. If the numbers are equal, the interface is satisfied. The type signature is provided by GoAL in an easily comparable form.

Worth noting is the usage of recursive common-table-expressions (CTEs). This feature is very new to SQLite, greatly simplifies this type of query, and already boasts very good performance. Despite its simple appearance, the query's execution performance is very robust. Query execution on a snapshot of the entire Go standard library from April 7th, 2013, consisting of 1779 structs and 170 interfaces took between 70-100ms on a Thinkpad T520 i7 laptop running Fedora 20 (time to create database not included).

Chapter 4

Implementation

4.1 Overview

As a problem-focused language, the GoAL DSL was defined entirely as a series of Lua functions that construct its AST, to avoid the need for complex parsing, and other tasks not directly related to the task of emitting information about a Go program.

Using the dynamic nature of Lua, the structure of the GoAL program is encoded via a series of nested Lua function calls. For Go, partial AST semantic annotation was achieved by using the go/types tool, a quasi-standard library developed by the Go development team, likely to be standardized in the future. Through this tool, type annotation was included for applicable nodes, thus enhancing the usability of the AST greatly for semantic analysis.

The GoAL system relies heavily on the usage of standard (and quasi-standard, as noted above) introspective features of the target language (Go). Firstly, the static introspective facilities of Go were used for parsing the Go source code and building the AST, as well as implementing the event-driven system used

to process it. Secondly, the dynamic introspective facilities of Go were used to provide communication between the bytecode runner component implemented in Go, and the bytecode compiler implemented in Lua.

Owing to the simplicity and syntactic familiarity of embedding Lua in Go using the 'luar' package, Lua was used both to construct the GoAL program representation in an embedded DSL, as well as to generate the bytecode, keeping the responsibilities of the Go runtime component small.

4.1.1 GoAL Bytecode Emission via Lua

Via standard bytecode compilation techniques, GoAL procedures are compiled to a custom bytecode specialized for extracting data from Go AST nodes. The bytecode compiler is implemented in Lua, and is quite small owing to code reuse permitted by Lua's dynamic nature, and boiler-plate-free reflection of symbols defined by the Go bytecode runner component.

Bytecode emission was achieved by operating on an AST representation formed from the Lua realization of the DSL. The bytecode emission provides optimizations such as common subobject optimization for repeated sub-expressions, taking account of knowledge of the immutability of the data on which the system is operating.

To emit bytecode for a given GoAL program, a two-pass bytecode compiler was used. On the first pass, constants (such as string literals) are identified and merged, as well as common sub-expressions, and bytecode locations are allocated and mostly filled. On the second pass, the fields of the bytecode requiring object references and bytecode locations are filled.

The Lua runtime component interacts with the Go component through the 'luar' package. This allows syntactically familiar, albeit somewhat inefficient,

access to Go objects from Lua. Dynamic Lua inspection was used to reduce code duplication, primarily in the form of automatic inference of GoAL DSL elements such as 'typeof' from the name of their runtime symbol (in this case, the symbol was named 'SMEMBER_typeof').

4.1.2 Evaluating a GoAL Analysis Pass

Analysis using the GoAL runtime is triggered through the 'Analyze' global Lua procedure. This procedure takes a set of package locations (directories or files), and operates on each Go file in the package.

Before analysis, a pre-analysis phase creates lookup tables for use with byte-code evaluation. This is done to speed up access to Go reflection by caching common lookups. Since the amount of names used by the go/ast API is finite, a code was attached to each name, and since the amount of types used by the go/ast API is finite, a lookup table of these codes was used to speed-up reflection.

During analysis, the following steps are taken for each file:

1. Parse the file using the go/parse package, obtaining a tree of AST nodes

2. Create a mapping from the AST nodes to the results of type inference, given by go/types

3. Set up an evaluation environment

While the Lua programming language is used to construct the GoAL program AST, the GoAL language runtime is centered around a stack. This stack operates on interface{} values (representing any type, in Go), wrapped with additional type information useful to GoAL.

4.1.3 GoAL Git Tool

Alongside GoAL, a tool implemented in Lua and Bash labelled the 'GoAL Git Tool' was developed. This tool integrates with git, storing a database in the .git folder of a repository, and running a GoAL program on every commit. GoAL, a dynamic generation scheme was used to translate the entire AST into a database-ready representation. To achieve this, the AST representation was processed into a stream of tuples, representing information for a given node.

References to other nodes were created using GoAL's 'id' operator, which is unique within a single pass of the GoAL system on an AST. To ensure a unique key for each object, a tag was also prefixed, tied to a commit, which uniquely marks each pass added to the database. [10]

For simplicity in formulating queries, the database was defined in terms of universal data tables 'node_data' and 'node_links'. Function declarations were special-cased, as they form the root of most analyses, and contain more information than the usual node.

Everything except methods and functions (FuncDecl instances) are stored in two tables, node_data and node_links. FuncDecl instances are stored in their own table with a fields for their receiver and name. The schema for node_data & node_links is defined as follows:

```
CREATE TABLE node_data (
        tag INTEGER, id INTEGER, kind TEXT,
        // Distinguishing label of node, for example a name, operator like '>=', etc.
        data TEXT,
        // Null if not an expression (ie, is a statement)
        location TEXT, type TEXT,
PRIMARY KEY (tag,id))

CREATE TABLE node_links (
```

```
        tag INTEGER, id INTEGER, label TEXT,
        // For code blocks, number from 0 onwards. Otherwise, nil.
        link_number INTEGER,
        //ID of linked-to node
        link_id INTEGER,
PRIMARY KEY (tag,id,label,link_number))
```

Using recursive queries, now supported in all major open-source relational databases, one can flexibly operate on this representation to perform arbitrary analyses.

4.1.4 Dependencies & Building

Bundled Dependencies

The LuaJIT engine was bundled with Go, as, though it is larger than the standard Lua distribution it provides much faster interpretation. The engine also (as the name suggests) boasts very sophisticated just-in-time (JIT) optimizations for long-running code. This ensures that the Lua compilation backend, which transforms GoAL expressions to executable bytecode, imposes as little overhead as possible.

The GoAL system is dependent on the following Go packages:

- github.com/aarzilli/golua/lua, for Lua embedding

- github.com/mattn/go-sqlite3, for SQLite integration

- github.com/lib/pq, for PostGres integration

- github.com/stevedonovan/luar, for accessing Go objects from Lua

- github.com/shavac/readline, for Lua interactive support (minor feature)

By bundling versions of these libraries and providing a convenient query language for uniform access, some issues of discoverability and stability are mitigated for the end-user. During the development of GoAL, two libraries (go-sqlite3 & go/types) were updated in API-breaking ways, forcing updating, and, in the case of go/types, reconsideration of design. A move to the most recent version of go/types proved to cause some issues, requiring modification of bundled libraries to work-around an API modification. The GoAL system provides the simplifications over direct AST access via Go, shielding the user from API flux.

External Dependencies

For compilation, the SQLite3 development package for the target operating system is required.

When dependencies are satisfied, testcases can be run with ./run.sh. For a complete example usage of the tool see "godb/isat1.lua" in the repository [1].

Chapter 5

Analysis

5.1 Evaluation of Interface Satisfaction

As described in the Design chapter, the results of applying an interface satisfaction query to the entire Go standard library was evaluated. The performance was found to be very acceptable for query solving. Query execution on a snapshot of the entire Go standard library from April 7th, 2013, consisting of 1779 structs and 170 interfaces took between 70-100ms on a Thinkpad T520 i7 laptop running Fedora 20. In contrast, creating the database for the information required 7 seconds, heavily bound by the insertion of data into the relational database with (selective) data for the entire Go standard library (nearly 300,000 lines of code).

5.2 Measurements Taken

In order to evaluate the performance and stability of the GoAL tool, a variety of Go repositories were tested. The first 3642 valid Go packages implemented as

git repositories were accessed (cloned) using go-search.org's [8] package-search API.

After cloning, the Git Go Tool was initialized. This was set then to emit the data for a single commit, essentially operating on every node in the AST. This was done twice, first using SQLite3 to obtain combined statistics for AST traversal & database emitting (hard to separate), and finally to produce statistics about AST traversal alone. Using this technique, the relative portions spent for each task can be seen. By and large, database emitting was the bottleneck. It is theorized that a more efficient storage pattern would help greatly.

In measurement gathering, only non-comment lines of code were considered. Wherever 'lines' or 'SLOC' are used, they are meant equivalently.

5.3 Measurement Visualization

Distribution of SLOC in repositories under analysis.

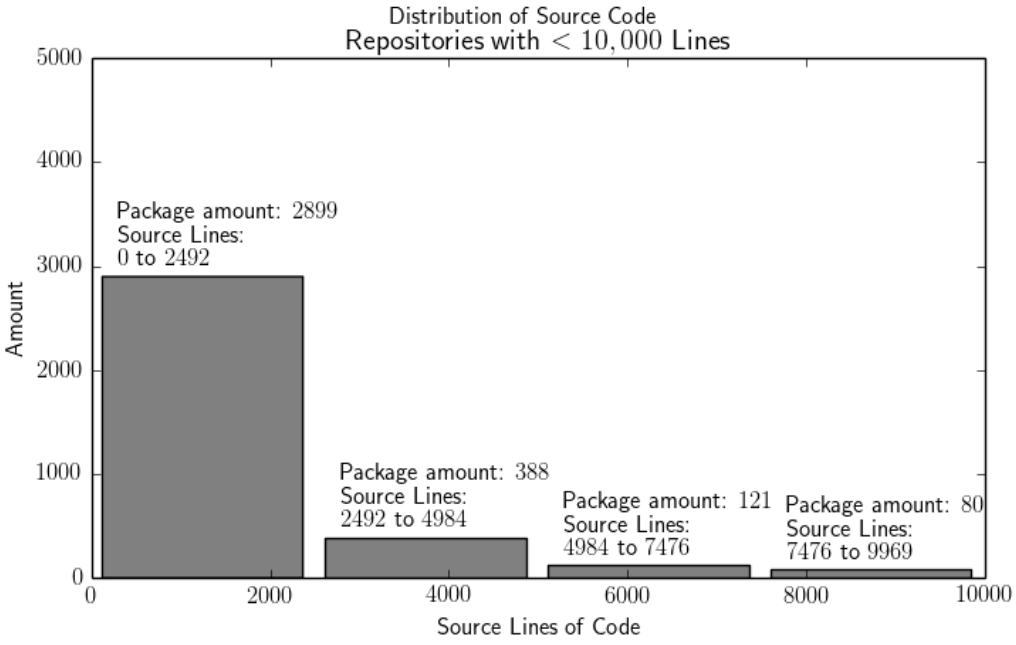

Figure 5.1: Repositories with Go source lines $< 10,000$

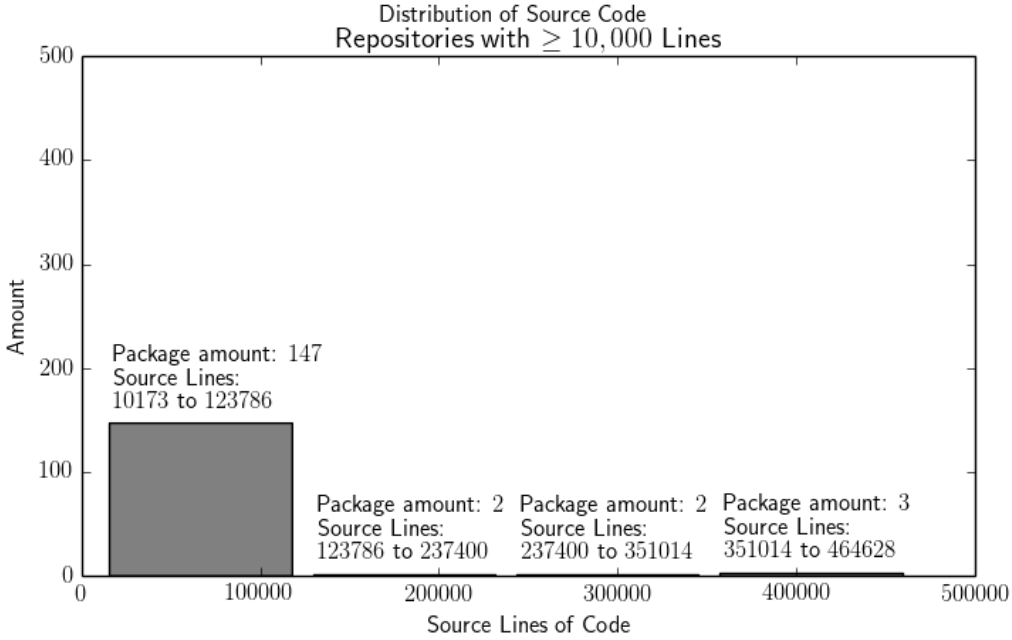

Figure 5.2: Repositories with Go source lines $\geq 10,000$

The first stage of the source-code analysis is reading every file from disk, and running a lexer to extract token used to construct an AST, using the go/parse package. This AST is then annotated with parsing provided by the go/types package. As these operations are seen as irreducible elements from the design standpoint of GoAL (the respective go/parse and go/types packages implementing them as such), and thus are measured together.

Graphs for parsing and type inference.

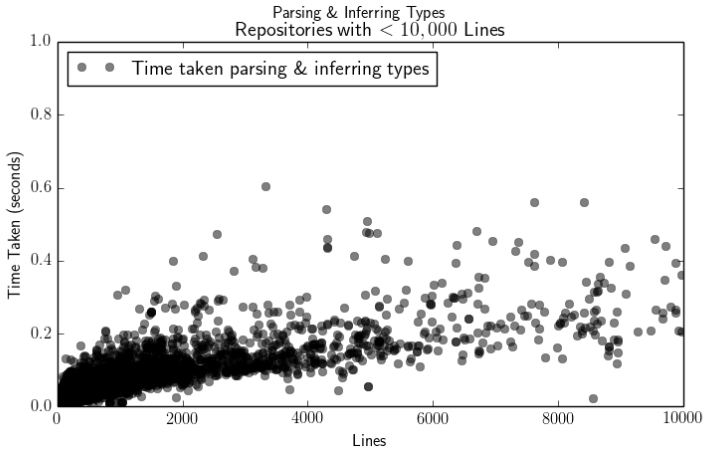

Figure 5.3: Source parsing & type inference performance, SLOC $< 10,000$

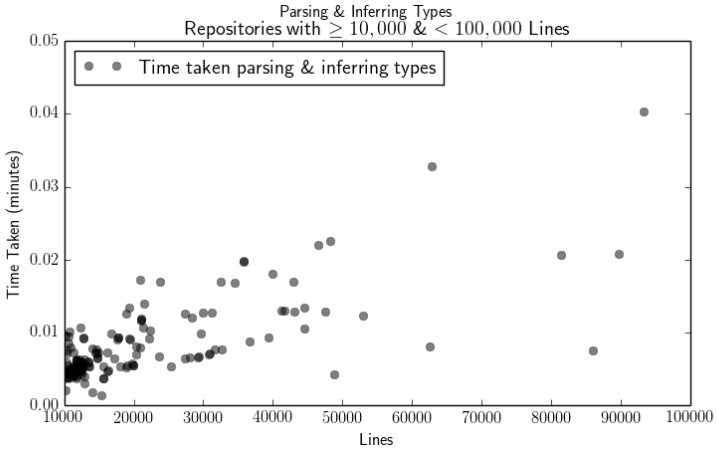

Figure 5.4: Source parsing & type inference performance, $10,000 \leq$ SLOC $< 100,000$

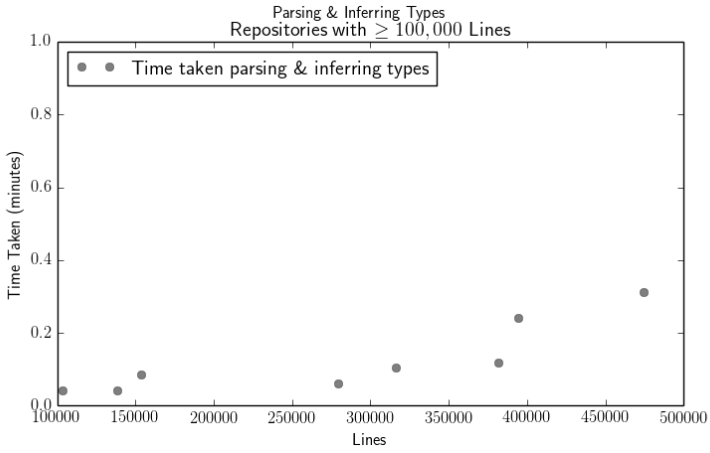

Figure 5.5: Source parsing & type inference performance, $100,000 \leq$ SLOC

Graphs for AST traversal & database row creation.

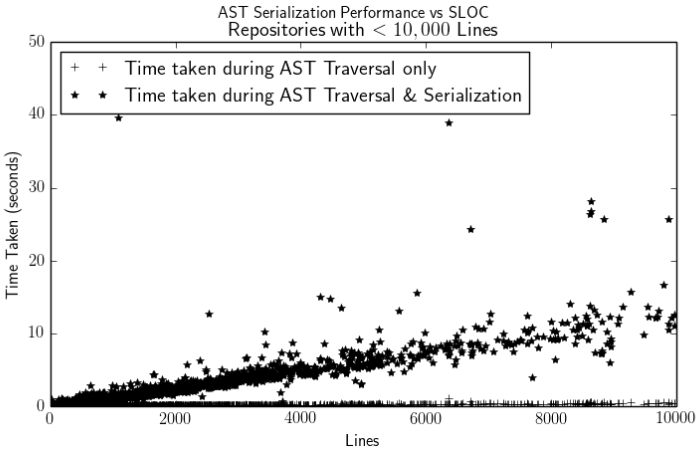

Figure 5.6: Data selection & serialization performance, SLOC $< 10,000$

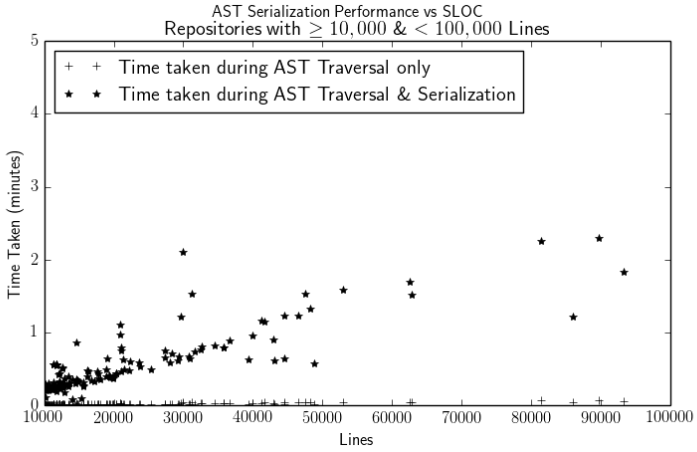

Figure 5.7: Data selection & serialization performance, $10,000 \leq$ SLOC $< 100,000$

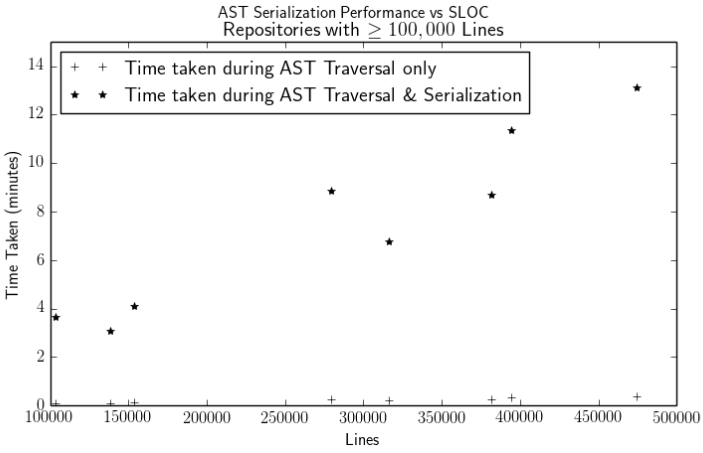

Figure 5.8: Data selection & serialization performance $100,000 \leq$ SLOC

Graph for AST traversal only.

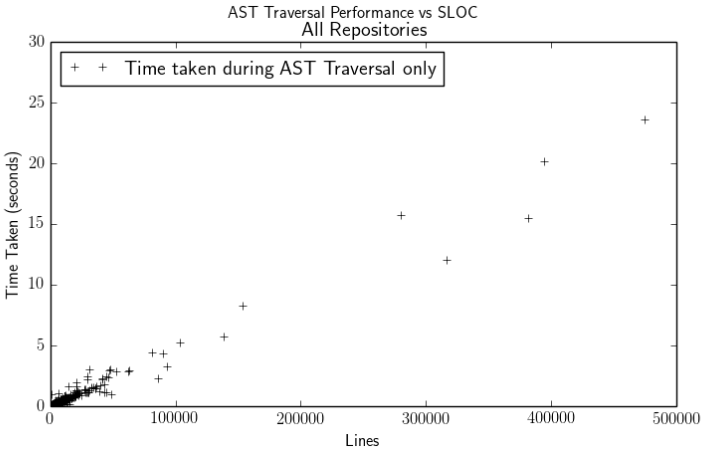

Figure 5.9: All repositories, AST traversal without data emission

5.4 Summary of Results

The main performance results can be summarized as follows:

Repositories with under 10,000 SLOC:

File reading, tokenization, parsing, type inference:

< 0.6 seconds

Running GoAL bytecode on each node of the AST (AST Traversal):

< 0.7 seconds

Running GoAL bytecode on each node of the AST & Emitting to SQLite3 (AST Traversal + Serialization):

< 40 seconds

Repositories with over 10,000 but under 100,000 SLOC:

File reading, tokenization, parsing, type inference:

< 2.4 seconds

Running GoAL bytecode on each node of the AST (AST Traversal):

< 4.2 seconds

Running GoAL bytecode on each node of the AST & Emitting to SQLite3 (AST Traversal + Serialization):

< 140 seconds

Repositories with over 100,000 but under 500,000 SLOC:

File reading, tokenization, parsing, type inference:

< 22 seconds

Running GoAL bytecode on each node of the AST (AST Traversal):

< 24 seconds

Running GoAL bytecode on each node of the AST & Emitting to SQLite3 (AST Traversal + Serialization):

< 780 seconds

Overall, the performance test was IO bound, in terms of writing to the SQL database. Quite a lot of data was written, encoding the entire AST. Performance tuning is likely to be most fruitful first optimizing data output to SQLite, and then parallelizing AST traversal. To operate at the level of performance required for most uses, incremental outputting of information is required, preventing this overhead being taken everytime a small change has been made.

Chapter 6

Conclusion

6.1 Identified Limitations & Shortcomings

A known issue in the implementation as of April 4th is that the 'typeof' operation does not return the full package path, causing a potential for name-clashing in queries. This was introduced during the move to go/types, and unfortunately was not resolved in time.

The GoAL system, while in theory multi-threadable without too hassle, is currently only single-threaded.

Additionally, a major source of semantic information was the go/types package (currently existing as "code.google.com/p/go.tools/go/types"). However, through the period of development, this library experienced a redesign. Initially, a source copy of go/types was forked from the in-development Go standard library on October 3rd, 2013. This was, however, moved out of the standard presumably due to API reconsideration. An effort was made to match upstream efforts February 25, 2014 which, facing uncertainty in the new design, lead to a hacked copy of the library in the source tree. The

update to the library was more selective about which nodes were annotated for efficiency reasons. However, due to complexity in using the old API for the initial purpose, two simple one-line modifications were made to the new code.google.com/p/go.tools/go/types library to mimic the old API. It is hoped that future library development of this library will provide the documentation that was felt missing in resolving this issue.

One limitation of the system is that, as implemented, go/types cannot handle relative paths within repositories. Since relative imports fail, the types used from them remain indeterminate. Nevertheless, the system can continue, providing partial results for anything not using relative imports.

6.2 Room for Improvement

While a number of features were considered for inclusion to GoAL, not all of them were realized.

An identified possible future direction for GoAL includes support for automatic incremental upkeep of databases, which would greatly increase the responsivity tools written using it, such as the GoAL Git Tool.

As well, builtin support for commit-tagging, and related optimizations such as diff-based compression would also greatly aid to the usefulness of the GoAL Git Tool, which scales poorly (linearly) with the number of commits made.

6.3 Closing Summary & Statement of License

A system for analyzing the Go programming language was implemented, deemed the GoAL system. Using the GoAL system, a scalable case study for solving

interface satisfaction was demonstrated. As well, a tool, deemed the GoAL Git Tool, which encodes the Go AST entirely, was developed. The performance of these tools was evaluated.

The GoAL System, discussed herein, is, with the explicit exception of the .libs/ & dependencies/ and any content not originating from the project, declared to be available under **public domain**.

The source code can be accessed via git by the URL https://github.com/ludamad/goquery.

Bibliography

[1] Adam Domurad (author), *GoAL system & GoAL git tool source.* http://github.com/ludamad/goquery

[2] Go development team, *The Go Programming Language Specification.* http://golang.org/ref/spec

[3] *How SQLite Is Tested* https://www.sqlite.org/testing.html

[4] *Go 1.0 is Released* http://blog.golang.org/go-version-1-is-released

[5] Go development team, *Go 1.3+ Compiler Overhaul.* https://-docs.google.com/document/d/1P3BLR31VA8cvLJLfMibSuT-dwTuF7WWLux71CYD0eeD8/preview?sle=true

[6] Mike Pall *LuaJIT FQA* http://luajit.org/faq.html

[7] Eclipse Foundation *About the Eclipse Indexer.* http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.cdt.doc.user%2Fconcepts%2Fcdt_c_indexer.htm

[8] *go-search.org.* http://go-search.org/

[9] *SQL Wikipedia Article.* http://en.wikipedia.org/wiki/SQL

[10] *Git Wikipedia Article.* http://en.wikipedia.org/wiki/Git_(software)

[11] *Hacker News, "SQLite now supports recursive CTEs"*. https://news.ycombinator.com/item?id=7438164