Bachelorarbeit

Dynamic Analysis of Message-Passing Go Programs

Erik Daniel Kassubek

Gutachter: Prof. Dr. Thiemann

Betreuer: Prof. Dr. Thiemann

Prof. Dr. Sulzmann

Albert-Ludwigs-Universität Freiburg

Technische Fakultät

Institut für Informatik

Lehrstuhl für Programmiersprachen

3. April 2017

Bearbeitungszeit

31. 10. 2022 – 31. 01. 2023

Gutachter

Prof. Dr. Thiemann

Betreuer

Prof. Dr. Thiemann

Prof. Dr. Sulzmann

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift

Zusammenfassung

(TODO: Zusammenfassung (Abstract) schreiben)

Inhaltsverzeichnis

1 Einführung

(TODO: Einführung schreiben)

2 Hintergrund

3 Tracer

Um ein Program analysieren zu können, wird der Ablauf eines Programmdurchlaufs aufgezeichnet. Dazu wurde ein Tracer implementiert, in welchem die Channel-Operations durch Drop-In Replacements ersetzt werden. Diese führen die eigentlichen Operationen aus und zeichnen gleichzeitig den Trace auf. Die Ersetzung durch die Drop-In Replacements kann dabei automatisch durch einen Instrumenter erfolgen, welche mit Hilfe des Abstract Syntax Tress die Ersetzungen vornimmt.
Die Idee einen Abstract Syntax Tree auf Go Programmen zu verwenden, um dadurch Funktionen in den Code zu integrieren durch welche ein Trace aufgezeichnet wird sind nicht neu. Beispiele dazu finden sich in [1] und [2].

(EXTEND: Tracer Einführung)

3.1 Trace

(DRAFT: Trace) Der Aufbau des Trace basiert auf [3]. Der Trace wird für jede Routine separat aufgezeichnet. Die Syntax des Traces in EBNF gibt sich folgendermaßen:

$$
\begin{array}{lcll}
T & = & '' \, [\, '', \, \{\, U \,\}, \, '' \,] \, '' ; & \text{Trace} \\
U & = & '' \, [\, '', \, \{\, t \,\}, \, '' \,] \, '' ; & \text{lokaler Trace} \\
t & = & signal(i) \mid wait(i) \mid pre(as) \mid post(i,j,a) & \text{Event} \\
 & & \mid post(default) \mid close(x); & \\
a & = & x, \; (\, '' \, ! \, '' \mid '' \, ? \, '' \,); & \\
as & = & a \mid (\{\, a \,\}, [\, '' \, default \, '' \,]); &
\end{array}
$$

wobei i die Id einer Routine, j einen Zeitstempel und x die Id eines Channels darstellt. Die Events haben dabei folgende Bedeutung:

- $signal(i)$: In der momentanen Routine wurde eine Fork-Operation ausgeführt, d.h. eine neue Routine mit Id i wurde erzeugt.
- $wait(i)$: Die momentane Routine mit Id i wurde soeben erzeugt. Dies ist in allen Routinen außer der Main-Routine das erste Event in ihrem lokalen Trace.
- $pre(as)$: Die Routine ist an einer Send- oder Receive-Operation eines Channels oder an einem Select-Statement angekommen, dieses wurde aber noch nicht ausgeführt. Das Argument as gibt dabei die Richtung und den Channel an. Ist $as = x!$, dann befindet sich der Trace vor einer Send-Operation, bei $as = x?$ vor einer Receive-Operation. Bei einem Select-Statement ist as eine Liste aller Channels für die es einen Case in dem Statement gibt. Besitzt das Statement einen Default-Case, wird dieser ebenfalls in diese List aufgenommen, also $as = [\, x_1?, \ldots, x_n?, default]$.
- $post(i,j,a)$: Dieses Event wird in dem Trace gespeichert, nachdem eine Send- oder Receive-Operation erfolgreich abgeschlossen wurde. i gibt dabei die Id und j den momentanen Zeitstempel der sendenden Routine an. a gibt wieder an, ob es sich um

eine Send- ($a = x!$) oder Receive-Operation ($a = x?$) auf dem Channel x handelt. Durch die Speicherung der Id und des Zeitstempels der sendenden Routine bei einer Receive-Operation lassen sich die Send- und Receive-Operationen eindeutig zueinander Zuordnen.

- $post(default)$: Wird in einem Select-Statement der Default-Case ausgeführt, wird dies in dem Trace der entsprechenden Routine durch $post(default)$ gespeichert.

- $close(x)$: Mit diesem Eintrag wird das schließen eines Channels x in dem Trace aufgezeichnet.

Man betrachte als Beispiel das folgende Programm in Go mit dem dazugehörigen Trace.

```
func main() { // routine 1
    x := make(chan int) // chan 1
    y := make(chan int) // chan 2
    a := make(chan int) // chan 3

    go func() { x <- 1 }()       // routine 2
    go func() { <-x; x <- 1 }() // routine 3
    go func() { y <- 1; <-x }() // routine 4
    go func() { <-y }()          // routine 5

    select {
    case <-a:
    default:
    }
}
```

Dieser ergibt den folgenden Trace:

$$[[signal(2)\ signal(3)\ signal(4)\ signal(5)\ pre(3?, default)\ post(default)]$$
$$[wait(2)\ pre(1!)\ post(2,1,1!)]$$
$$[wait(3)\ pre(1?)\ post(2,1,1?)\ pre(1!)\ post(3,2,1!)]$$
$$[wait(4)\ pre(2!)\ post(4,1,2!)\ pre(1?)\ post(3,2,1?)]$$
$$[wait(5)\ pre(2?)\ post(4,1,2?)]]$$

Aus diesem lässt sich der Ablauf der einzelnen Operationen auf den Channels eindeutig nachverfolgen.

3.2 Tracer

(DRAFT: Tracer) Um die beschriebenen Traces erzeugen zu können wird ein Tracer verwendet. Eine Implementierung des Tracers findet sich in [4]. Dabei werden Channel-Operationen in einem Go Programm durch spezielle Funktionen ersetzt, welche sowohl die eigentliche Operation ausführen, als auch die Aufzeichnung der Operationen in dem Trace vornehmen. Die Channels selber werden durch Structs *Chan* ersetzt, welche neben dem eigentlichen Channel *ch* die Id des Channels, sowie Variablen zur Speicherung der Sender Id (*sender*) und des Zeitstempels (*ts*) von Send-Operationen enthält. Zusätzlich werden einige globale Variablen gespeichert. Dabei handelte es sich um

- *nr*: Anzahl der Routinen, welche bereits erzeugt wurden
- *nc*: Anzahl der Channels, welche bereits erzeugen wurden
- *traces*: Object zur Speicherung der Traces
- *counter*: Object zur Speicherung der Timestamps für die Routinen

Im folgenden Bezeichne $trace_i$ den Trace der i-ten Routine und $counter_i$ den Zeitstempel der i-ten Routine. c bezeichnet den Index der Routine, von welcher eine Operation ausgeführt wird.

Die folgende Auflistung zeigt welche Operationen ersetzt werden, und was durch diese Ersetzung geschieht. $traces_i + [\ Events\ ]$ bezeichne dabei, dass $Events$ in $traces_i$ eingefügt wird. *spawn* p beschreibt die Erzeugung einer neuen Routine und Ausführung von p auf dieser Routine.

$$
\begin{array}{lcl}
instr(a : p) & = & instr(a) : instr(p) \\
instr(x := make(chan\ i, s)) & = & x := Chan\{c : make(chani, s),\ id : nc\}; \\
 & & nc = nc + 1; \\
instr(x := make(chan\ i)) & = & instr(x := make(chan\ i, 0)) \\
instr(spawn\ p) & = & nr = nr + 1;\ trace_c + [\ spawn(nr)\ ]; \\
 & & spawn\ instr(p);\ trace_{nr} + [\ wait(nr)\ ] \\
instr(x \leftarrow i) & = & counter_c = counter_c + 1;\ trace_c + [\ pre(x.id!)\ ]; \\
 & & ch.sender.push(c);\ ch.ts.push(counter_c); \\
 & & x.ch \leftarrow i;\ trace_c + [\ post(c,\ counter_c,\ x.id!)\ ] \\
instr(\leftarrow x) & = & counter_c = counter_c + 1;\ trace_c + [\ pre(x.id?)\ ]; \\
 & & \leftarrow x.ch; \\
 & & trace_c + [\ post(x.sender.pop(),\ x.ts.pop(),\ x.id?)\ ]; \\
instr(v :=\leftarrow x) & = & counter_c = counter_c + 1;\ trace_c + [\ pre(x.id?)\ ]; \\
 & & temp :=\leftarrow x.ch; \\
 & & trace_c + [\ post(x.sender.pop(),\ x.ts.pop(),\ x.id?)\ ]; \\
 & & v = temp; \\
instr(close(x)) & = & close(x.ch);\ trace_c + [\ close(x.id)\ ]; \\
instr(select\ \{case\ (v :=\leftarrow x_i & & \\
|\ \leftarrow x_i) \Rightarrow p_i\}_{i\in\{1,\ldots,n\}}) & = & trace_c + [\ pre(x_1.id, \ldots, x_n.id)\ ]; \\
 & & select\ \{case(instr(v :=\leftarrow x_i) \mid instr(\leftarrow x_i)) \\
 & & \Rightarrow (trace_c + [post(x_i.sender.pop(),\ x_i.ts.pop(),\ x_i.id?)]; \\
 & & instr(p_i))\}_{i\in\{1,\ldots,n\}} \\
instr(select\ \{(case\ (v :=\leftarrow x_i & & \\
|\ \leftarrow x_i) \Rightarrow p_i) \mid p\}_{i\in\{1,\ldots,n\}}) & = & trace_c + [\ pre(x_1.id, \ldots, x_n.id)\ ]; \\
 & & select\ \{(case(instr(v :=\leftarrow x_i) \mid instr(\leftarrow x_i)) \\
 & & \Rightarrow (trace_c + [post(x_i.sender.pop(),\ x_i.ts.pop(),\ x_i.id?)]; \\
 & & instr(p_i)))|\ trace_c + [post(default)];\ instr(p)\}_{i\in\{1,\ldots,n\}} \\
\text{Für alle anderen } a: \ instr(a) & = & a
\end{array}
$$

(EXTEND: Tracer)

3.3 Instrumenter

(DRAFT: Instrumenter) Um den Trace zu erzeugen, müssen verschiedene Operationen durch Funktionen des Tracers ersetzt werden. Soll der Tracer auf ein Programm angewendet werden, welches bereits ohne ihn implementiert worden ist, bedeutet dies einen signifikanten Arbeitsaufwand. Da sich der Tracer auch negativ auf die Laufzeit des Programms auswirken kann, ist es in vielen Situationen nicht erwünscht, ihn in den eigentlichen Release-Code einzubauen, sondern eher in eine eigenständige Implementierung, welche nur für den Tracer verwendet werden. Um dies zu automatisieren wurde ein zusätzliches Programm implementiert, welches in der Lage ist, den Tracer in normalen Go-Code einzufügen. Die Implementierung, welche ebenfalls in [4] zur Verfügung steht arbeitet dabei mit einem Abstract Syntax Tree. Bei dem Durchlaufen dieses Baums werden die entsprechenden Operationen in dem Programm erkannt, und durch ihre entsprechenden Tracer-Funktionen ersetzt bzw. ergänzt.

3.4 Laufzeit / Overhead

(TODO: Laufzeit)

4 Zusammenfassung

(TODO: Zusammenfassung schreiben)

5 Acknowledgments

(TODO: Acknowledgments schreiben)

Abbildungsverzeichnis

Tabellenverzeichnis

ToDo Counters

(TODO: Remove ToDo Counter List)

To Dos: 6; 1, 2, 3, 4, 5, 6

Parts to extend: 2; 1, 2

Draft parts: 3; 1, 2, 3

Literaturverzeichnis

[1] S. Taheri and G. Gopalakrishnan, "Automated Dynamic Concurrency Analysis for Go," *arXiv e-prints*, p. arXiv:2105.11064, May 2021.

[2] I. Danyliuk, "Visualizing concurrency in go," 2016. `https://divan.dev/posts/go_concurrency_visualize/`.

[3] M. Sulzmann and K. Stadtmüller, "Two-phase dynamic analysis of message-passing go programs based on vector clocks," *CoRR*, vol. abs/1807.03585, 2018.

[4] E. Kassubek, "GoChan." `https://github.com/ErikKassubek/GoChan`, 2022. v 0.1.