Liveness-Driven Random Program Generation

Gergö Barany
Inria Paris, France
gergo.barany@inria.fr

LOPSTR 2017, Namur, Belgium

Motivation

Context: automatic tool for finding *missed optimizations*

Generated source code:

```
int f(int p, int q) {
  return q + (p % 6) / 9;
}
```

($p \% 6 \in [-5, 5]$,
division truncates to 0)
recently fixed

Clang:

```
movw r2, #43691
movt r2, #10922
smmul r2, r0, r2
add r2, r2, r2, lsr #31
add r2, r2, r2, lsl #1
sub r0, r0, r2, lsl #1
movw r2, #36409
movt r2, #14563
smmul r0, r0, r2
asr r2, r0, #1
add r0, r2, r0, lsr #31
add r0, r0, r1
bx lr
```

GCC:

```
mov r0, r1
bx lr
```

Randomized differential testing

popularized by Yang et al., “Finding and understanding bugs in C compilers”, PLDI ’11

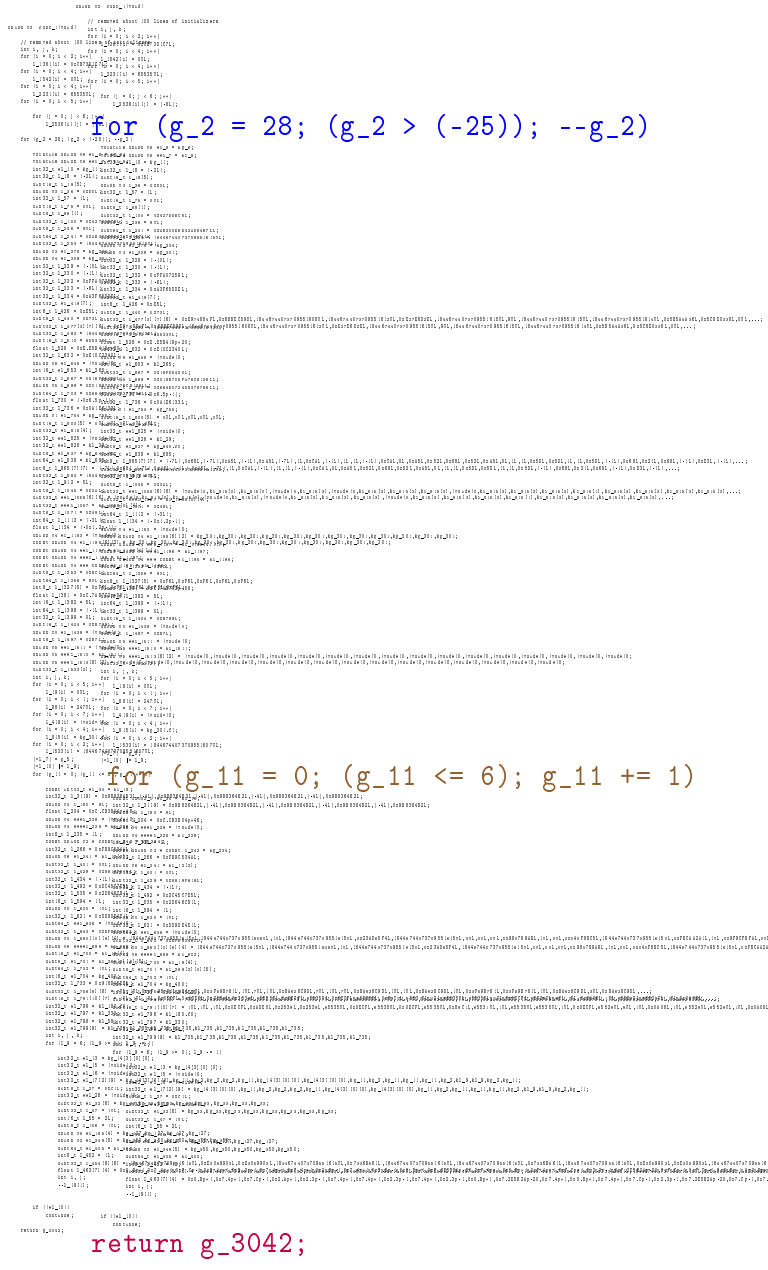
How to find compiler bugs:

- generate random source code
- compile with different compilers
- compare binaries (code or behavior)

Csmith: standard C program generator, has found hundreds of bugs

this work: ldrgen, new random C code generator

A problem with Csmith



162 LOC (without 100 lines of inits)
compiles to 8 instructions:

```
func_1:
    movl $27, %eax
.L2:
    movl %eax, g_2(%rip)
    subl $1, %eax
    movl $7, g_11(%rip)
    cmpl $-26, %eax
    jne .L2
    movl g_3042(%rip), %eax
    ret
```

vast majority of code is dead code

Liveness and dead code

live variable: variable that may be used in the future
dead variable: variable that is definitely not used in the future

```
x = y + z;
return x;
```

`x` live after assignment

```
x = y + z;
return y;
```

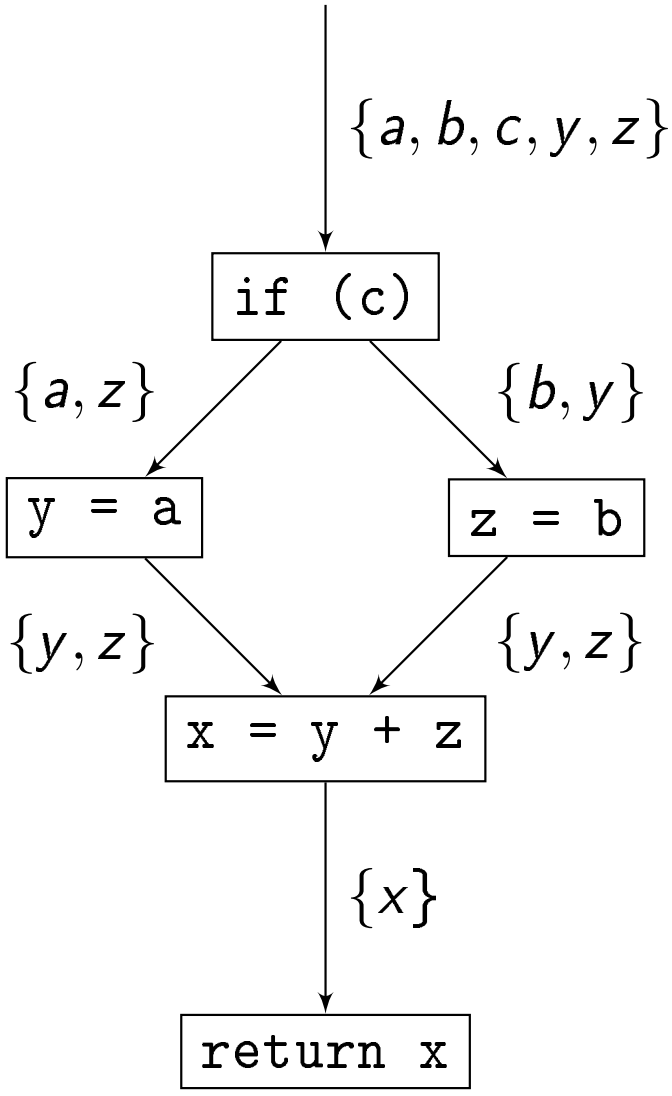
`x` dead after assignment

by extension: live code computes value maybe used in future
dead code elimination: standard compiler pass, removes dead code

Dead $\neq$ unreachable: dead code more than `if (false) ...`

Live variable analysis

backwards data flow analysis



live-in set $S^\bullet$: live before S
live-out set S°: live after S

transfer function for $\mathrm{v} = \mathrm{e}$:

$$S^\bullet = (S^\circ \setminus \{v\}) \cup FV(e)$$

control flow split:

$$S^\circ = \bigcup_{S_i \in succ(S)} S_i^\bullet$$

compute least fixed point

Generation of fully live programs

want to generate fully live programs: all statements are live

idea: perform liveness analysis during (backwards!) generation

but I am lazy:

- do not want to generate a control flow graph
- do not want to backtrack/iterate to fixed point

long-forgotten idea:

structural data-flow analysis

Structural (full) liveness analysis

liveness triple: $\langle S^\bullet \rangle \;\; S \;\; \langle S^\circ \rangle$

$$\text{Assign}\ \frac{v \in S^\circ \qquad S^\bullet = (S^\circ \setminus \{v\}) \cup FV(e)}{\langle S^\bullet \rangle \;\; v = e \;\; \langle S^\circ \rangle}$$

$$\text{Sequence}\ \frac{\langle S_1^\bullet \rangle \;\; S_1 \;\; \langle S_2^\bullet \rangle \qquad \langle S_2^\bullet \rangle \;\; S_2 \;\; \langle S_2^\circ \rangle \qquad S_2^\bullet \neq \emptyset}{\langle S_1^\bullet \rangle \;\; S_1 \;;\; S_2 \;\; \langle S_2^\circ \rangle}$$

$$\text{If}\ \frac{\langle S_1^\bullet \rangle \;\; S_1 \;\; \langle S^\circ \rangle \qquad \langle S_2^\bullet \rangle \;\; S_2 \;\; \langle S^\circ \rangle \qquad S^\bullet = S_1^\bullet \cup S_2^\bullet \cup FV(c)}{\langle S^\bullet \rangle \;\; \texttt{if}\ (c)\ S_1\ \texttt{else}\ S_2 \;\; \langle S^\circ \rangle}$$

side conditions to ensure full liveness

program S fully live iff $\langle S^\bullet \rangle \;\; S \;\; \langle \emptyset \rangle$ derivable

Example: Failed derivation

Programs with dead code cannot be proved fully live:

$$\text{Sequence}\ \frac{\text{Assign}\ \dfrac{\mathtt{x} \notin \{\mathtt{y}\}\ \lightning}{\langle S^\bullet \rangle\ \mathtt{x = y + z}\ \langle \{\mathtt{y}\} \rangle} \qquad \langle \{\mathtt{y}\} \rangle\ \mathtt{return\ y}\ \langle \emptyset \rangle}{\langle S^\bullet \rangle\ \mathtt{x = y + z; return\ y}\ \langle \emptyset \rangle}$$

Analyzing loops

$$\text{While}\quad \frac{\langle B^\bullet\rangle\ B\ \langle B^\circ\rangle \qquad L^\bullet = B^\bullet \cup L^\circ \qquad L^\circ \neq \emptyset \qquad B^\circ = L^\circ \cup B^\bullet \cup FV(c)\ \text{(minimal)}}{\langle L^\bullet\rangle\ \texttt{while}\ (c)\ B\ \langle L^\circ\rangle}$$

Not constructive: How to compute the fixed point?

Not a problem for random generation: chooose least fixed point.

Example derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle\{\texttt{a},\texttt{b},\texttt{n}\}\rangle\ \texttt{n = n - 1}\ \langle\{\texttt{a},\texttt{b},\texttt{n}\}\rangle}}{\langle\{\texttt{a},\texttt{n},\texttt{t}\}\rangle\ \texttt{b = t}\ \langle\{\texttt{a},\texttt{b},\texttt{n}\}\rangle}\ \vdots}{\langle\{\texttt{b},\texttt{n},\texttt{t}\}\rangle\ \texttt{a = b}\ \langle\{\texttt{a},\texttt{n},\texttt{t}\}\rangle}\ \vdots}{\langle\{\texttt{a},\texttt{b},\texttt{n}\}\rangle\ \texttt{t = a + b}\ \langle\{\texttt{b},\texttt{n},\texttt{t}\}\rangle}\ \vdots}{\langle\{\texttt{a},\texttt{b},\texttt{n}\}\rangle\ \texttt{t = a + b; a = b; b = t; n = n - 1}\ \langle\{\texttt{a},\texttt{b},\texttt{n}\}\rangle}}{\langle\{\texttt{a},\texttt{b},\texttt{n}\}\rangle\ \texttt{while (n>0) \{ t=a+b; a=b; b=t; n=n-1 \}}\ \langle\{\texttt{a}\}\rangle}}{\vdots}}{\langle\{\texttt{n}\}\rangle\ \texttt{a=0;b=1;while(n>0)\{t=a+b;a=b;b=t;n=n-1\};returna}\ \langle\emptyset\rangle}$$

From structural analysis to code generation (1/2)

$$\text{Assign}\ \frac{v \in S^\circ \qquad S^\bullet = (S^\circ \setminus \{v\}) \cup FV(e)}{\langle S^\bullet \rangle\ \ v = e\ \ \langle S^\circ \rangle}$$

```
let assignment S° =
    let v = random_select S° in
    let e = random_expression () in
    ("v = e", (S° \ {v}) ∪ FV(e))
```

From structural analysis to code generation (2/2)

$$\text{If}\ \frac{\langle S_1^\bullet \rangle\ S_1\ \langle S^\circ \rangle \qquad \langle S_2^\bullet \rangle\ S_2\ \langle S^\circ \rangle \qquad S^\bullet = S_1^\bullet \cup S_2^\bullet \cup FV(c)}{\langle S^\bullet \rangle\ \texttt{if}\ (c)\ S_1\ \texttt{else}\ S_2\ \langle S^\circ \rangle}$$

let branch S° =
  let (t, $S_1^\bullet$) = random_statements S° **in**
  let (f, $S_2^\bullet$) = random_statements S° **in**
  let c = random_expression () **in**
  ("`if (c) t else f`", $S_1^\bullet \cup S_2^\bullet \cup FV(c)$)

Generation of loops

$$\text{While}\quad \frac{\langle B^\bullet\rangle\ B\ \langle B^\circ\rangle \qquad L^\bullet = B^\bullet \cup L^\circ \qquad L^\circ \neq \emptyset \qquad B^\circ = L^\circ \cup B^\bullet \cup FV(c)\ \text{(minimal)}}{\langle L^\bullet\rangle\ \texttt{while}\ (c)\ B\ \langle L^\circ\rangle}$$

Idea:

- generate random variable set B', condition c
- assume least fixed point $B^\circ = L^\circ \cup B' \cup FV(c)$
- generate body given B°
- add statements to ensure all $v \in B'$ used in body and live-in

Implementation (1/2)

Plugin for Frama-C analysis platform
Generator: about 600 lines of OCaml
Features:

- arithmetic, simple arrays and pointers
- `if`, `while`
- `for` loops implementing map-reduce on arrays:

```
v = ...;
for (unsigned int i = 0; i < N; i++) {
    v = v ∘ f(arr[i]);
}
```

- many flags for customization:

```
-fp -fp-only -int-only -bitwise -div-mod
-expr-depth -stmt-depth -block-length -loops
-max-args -max-live
```

Implementation (2/2)

Limitations:

- no `struct` (coming at some point)
- very limited use of pointers, no pointer arithmetic
- strictly structured code only (no `break`, `continue`, `goto`)
- a single function, no `main` function

Csmith is much, much more general.

Evaluation

1000 programs generated each

	generator	min	median	max	total
lines of code	Csmith	25	368.5	2953	459021
	`ldrgen`	12	411.5	1003	389939
instructions	Csmith	1	15.0	1006	45606
	`ldrgen`	1	952.5	4420	1063503
unique opcodes	Csmith	1	8	74	146
	`ldrgen`	1	95	124	204

generator	time (sec)	lines/sec	instrs/sec
Csmith	871	527	52.4
`ldrgen`	124	3140	8562.8

Summary

- random program generation for testing compiler optimizations
- fully live programs by structural analysis during generation
- much more effective than Csmith for this use case

`https://github.com/gergo-/ldrgen`

Thank you for your attention

This research was partially supported by ITEA 3 project no. 14014, ASSUME.