Goblint: Path-sensitive Data Race Analysis

Vesal Vojdani Varmo Vene

University of Tartu

Introduction

What is Goblint?

- General
  - A general analysis framework
- O'Caml
  - Analyses are specified in Objective Caml
  - (But it analyzes C code)
- Brogram
  - Means "program" in Persian dialect of Estonian . . .
- Linter
  - Such as splint

Introduction

What is Goblint?

- Goblint is *a static analyzer for Posix-threaded C*
- Focused on detecting multiple access *data races*
- Integrates with *Eclipse* C develpment environment
- Aims to be *sound* (ie. must detect all errors, but may give false alarms)
- Aims to be *efficient* enough to be able to analyze medium-to-large scale programs ($\geq$ 100 kLOC)
- Aims to be *precise* enough to be able to analyze medium-to-large scale programs ($\geq$ 100 kLOC)

Introduction

Main conflicts

- Soundness vs. C
- Efficiency vs. Precision

Introduction

Main conflicts

- Soundness vs. C
- Efficiency vs. Precision

Soundness vs. C

Restrict to the ”safe” subset of C:

- no `setjmp` and `getjmp`;
- no dynamic data structures;
- no recursion;
- . . .

Introduction

Main conflicts

- Soundness vs. C
- Efficiency vs. Precision

Soundness vs. C

Restrict to the "safe" subset of C:

- no `setjmp` and `getjmp`;
- no dynamic data structures;
- no recursion;
- . . .

Not as bad as it looks:

- we can still handle these constructs,
- but do not guarantee the soundness.

Introduction

Main conflicts

- Soundness vs. C
- Efficiency vs. Precision

Efficiency vs. Precision

We adopt normal data flow analysis techniques, but

- use functional approach to distinguish calling contexts,
- use dynamically adjustable path-sensitive analysis;
- use global invariant based concurrent analysis.

General framework

Stages of the analysis

- Transform the program to CFG
- Transform CFG to a constraint system
- Solve the constraint system

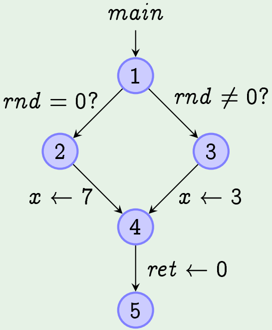
General framework

Stages of the analysis

- Transform the program to CFG
- Transform CFG to a constraint system
- Solve the constraint system

Example

```
int main () {
  int rnd;
  int x;
  if (rnd)
    x = 3;
  else
    x = 7;
  return 0;
}
```

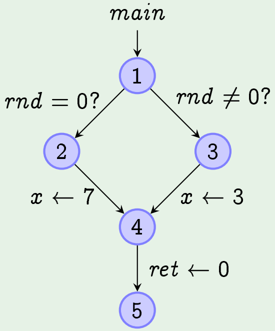


General framework

Stages of the analysis

- Transform the program to CFG
- Transform CFG to a constraint system
- Solve the constraint system

Example



$$\begin{array}{lcl} X_1 & \sqsupseteq & \mathcal{R}_0(\top) \\ X_2 & \sqsupseteq & \mathcal{R}_1(X_1) \\ X_3 & \sqsupseteq & \mathcal{R}_2(X_1) \\ X_4 & \sqsupseteq & \mathcal{R}_3(X_2) \bigsqcup \mathcal{R}_4(X_3) \\ X_5 & \sqsupseteq & \mathcal{R}_5(X_4) \end{array}$$

General framework

Stages of the analysis

- Transform the program to CFG
- Transform CFG to a constraint system
- Solve the constraint system

Example

$$\begin{array}{lcl} X_1 & \sqsupseteq & \mathcal{R}_0(\top) \\ X_2 & \sqsupseteq & \mathcal{R}_1(X_1) \\ X_3 & \sqsupseteq & \mathcal{R}_2(X_1) \\ X_4 & \sqsupseteq & \mathcal{R}_3(X_2) \bigsqcup \mathcal{R}_4(X_3) \\ X_5 & \sqsupseteq & \mathcal{R}_5(X_4) \end{array}$$

$$\begin{array}{lcl} X_1 & = & \mathcal{S}_1 \\ X_2 & = & \mathcal{S}_2 \\ X_3 & = & \mathcal{S}_3 \\ X_4 & = & \mathcal{S}_4 \\ X_5 & = & \mathcal{S}_5 \end{array}$$

General framework

Simplified Constraint System

$n \in \mathsf{N}$	(nodes of CFG)
$d \in \mathbb{D}$	(abstract program states)
$\langle n, d \rangle \in \mathsf{V} = \mathsf{N} \times \mathbb{D}$	(variables of constraint system)
$\mathcal{R} : (\mathsf{V} \to \mathbb{D}) \to \mathbb{D}$	(transfer functions = RHS-s of CS)

Note

- The system is infinite!!
- It can be (partially) solved using demand-driven solvers.
- (Fecht & Seidl, 1999)

Context-sensitivity

Example

```
void safeInc(int *v, pthread_mutex *m) {
  pthread_mutex_lock(m);
  v++;
  pthread_mutex_unlock(m);
}
```

Context-sensitivity

Example

```
void safeInc(int *v, pthread_mutex *m) {
  pthread_mutex_lock(m);
  v++;
  pthread_mutex_unlock(m);
}
```

Functional approach to interprocedural analysis

$f \in \mathcal{F}$	(function names)
$\mathsf{V} = \{\mathsf{N} \cup \mathcal{F}\} \times \mathbb{D}$	(variables of constraint system)

- A variable $\langle f, d \rangle$ denotes function call together with the entry state.
- NB! Does not behave well with recursion!

Path-sensitivity

man gcc on “-Wuninitialized”

These warnings are made optional because GCC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error . . .

Here is another common case:

```
int save_y;
if (change_y) save_y = y, y = new_y;
...
if (change_y) y = save_y;
```

This has no bug because "save_y" is used only if it is set.

Path-sensitivity

Example

```
int save_y;
if (change_y) save_y = y, y = new_y;
...
if (change_y) y = save_y;
```

What is the problem?

- There are 4 potential execution paths.
- Only 2 are logically possible.
- We need to distinguish execution paths.
- In general, there are an infinite number of paths!

Path-sensitivity

Example

```
int save_y;
if (change_y) save_y = y, y = new_y;
...
if (change_y) y = save_y;
```

Possible solution

- Instead of states use their powersets, but these might be infinite.
- One could try to use powersets with fixed maximum cardinality, but this is not only ugly but also very inefficient!

Path-sensitivity

Example

```
int save_y;
if (change_y) save_y = y, y = new_y;
...
if (change_y) y = save_y;
```

Our solution

- We only track the paths that are relevant to the analysis result.
- In this example, paths are relevant when the set of uninitialized variables are different.
- In general, relevance depends on the user-analysis. . .

Path-sensitivity

Dynamically adjustable path-sensitivity

$\mathbb{D}_b$ (abstract base state)

$\mathbb{D}_l$ (abstract user state)

$\mathbb{D} = \mathbb{D}_l \to \mathbb{D}_b$ (abstract state)

- We implement this as a power domain $\mathbb{D} = \mathcal{P}(\mathbb{D}_b \times \mathbb{D}_l)$, where the least upper bound merges the first components for identical states of the second.
- Note: if user domain $\mathbb{D}_l$ is finite, the $\mathbb{D}$ is also finite.

Concurrent Analysis

State explosion

- Precise concurrent analysis leads to state explosion.
- Eg. if there are two threads with 10 instructions each, then there are 184756 possible interleavings!

Concurrent Analysis

State explosion

- Precise concurrent analysis leads to state explosion.
- Eg. if there are two threads with 10 instructions each, then there are 184756 possible interleavings!

Global invariant based concurrent analysis

- Separate shared (ie. global) and local variables.
- Compute a single invariant for global state.
- Essentially, join all possible values in all program points.
- Now all threads can be analyzed sequentially.
- Very imprecise for base domain, but works well with user domains like lock-sets.
- Variant: compute the invariant only after the creation of the first thread.
- (Seidl & Vene & Müller Olm, 2003).

Experimental results

Small open source benchmarks

aget — a wget clone
pfscan — a parallel file scanner
knot — a web server
ctrace — a sample program of ctrace library
smtprc — a mail relay scanner

Benchmark	Size (kloc)	Time (s)	Warnings	Unguarded
aget	1.2	0.2	5	3
pfscan	1.3	0.4	0	0 (1)
knot	1.3	0.2	4	4 (6)
ctrace	1.4	0.3	2	0
smtproc	5.7	7.8	4	0

Conclusions

Ongoing and further works

- Equality analysis of addresses (with H.Seidl);
- Scalability improvements;
- Adding new analyses (eg. variable initialization, open-use-close analysis, etc.);
- Better handling of external functions;
- . . .

Additional information

- Goblint has an Open Source license
- You can download it from web: `http://goblin.at.mt.ut.ee/goblint/tracker/`