Completely genereic as-path-sensitive-as-necessary multithreaded API analysis

Vesal Vojdani

Department of Computer Science
University of Tartu

January 26, 2006

Introduction to the Goblint

- The goblint is a static analyzer for posix threaded C.
- It was scheduled to be released with Windows Vista. . .
  but has been delayed.
- Focused on detecting multiple access data races.
- Precise race detection requires detailed analysis of program flow
  - Resolving pointers and ambiguous function calls.
  - Distinguishing relevant paths within a function.
- We have detailed static traces of program.
- We want to let you use it.

Generic API analysis

- In the sense of "Checking system rules using system-specific programmer-written compiler extensions." (Engler et al, 2000)
  - Access to global variables require locking mutexes.
  - Disabled interrupts must be re-enabled.
  - Varialbes should be sanitized before use.
  - $\forall h : \mathsf{fopen}(h) \ldots \mathsf{fwrite}(.., h) \ldots \mathsf{fclose}(h)$
- Expose the program trace to the user to define temporal properties.
- Do it in a generic way: there should be no change to the base analysis when checking for a new property.

Specification language

- Not temporal logic. . .
- A bunch of DSLs hosted by O'Caml.
  - Domain language hosted by the module system.
  - Transfer functions hosted by the core language.
  - Analysis transformers hosted by the module system.
  - Analysis patterns hosted by SNOCOs.
- Goal: user can implement all the promised analyses in a few lines of code.
- The first two are fairly standard, so we'll look at transformers.
- Patterns are not implemented.

Analysis transformers

- They are essential to being “completely generic”.
- Implemented by functors.
- The context sensitive composition functor
  - Given a base analysis (constant propagation and points-to analysis)
  - And a user analysis (e.g. mutex analysis)
  - Produces a context sensitive user analysis (e.g. Goblint’s data race analysis)
- The path sensitive composition functor.
  - Context sensitive like above.
  - Adds path sensitivity (topic of next slide).
  - This one is our favourite.

Path sensitivity

man gcc on “-Wunitialized”

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.

What is the problem?

Example

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

- There are 4 potential execution paths.
- Only 2 are logically possible.
- We need to distinguish execution paths, but they can be inifinte!

As path sensitive as necessary

Example

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

- We only track the relevant paths.
- Paths are relevant when the set of uninitialized variables are different.
- Path sensitivity depends on the user-analysis, so how do we make it generic?

Defining the domain

- Let $\mathbb{D}_\mathtt{b}$ denote the domain of our base analysis and $\mathbb{D}_\mathtt{u}$ the user's domain.
- The needlessly path sensitive approach: $\mathbb{D}_\mathtt{b} \to \mathbb{D}_\mathtt{u}$.
- The as-path-sensitive-as-necessary domain: $\mathbb{D}_\mathtt{u} \to \mathbb{D}_\mathtt{b}$.
- We implement this as a power domain $\mathcal{P}(\mathbb{D}_\mathtt{b} \times \mathbb{D}_\mathtt{u})$, where the least upper bound merges the first components for identical states of the second.

Example

For the previous example we have after the first branch (assuming $\mathtt{y}$ is already initialized):

$$\{([\mathtt{change\_y} \mapsto 0], \{\mathtt{y}\}), ([\mathtt{change\_y} \mapsto \overline{\{0\}}], \{\mathtt{save_y}, \mathtt{y}\})\}$$

Defining the transfer functions

- We need to combine
  - $tf_{\mathrm{b}} \colon \mathbb{D}_{\mathrm{b}} \to \mathbb{D}_{\mathrm{b}}$
  - $tf_{\mathrm{u}} \colon \mathbb{D}_{\mathrm{u}} \to \mathbb{D}_{\mathrm{u}}$
- Into a function $tf \colon \mathcal{P}(\mathbb{D}_{\mathrm{b}} \times \mathbb{D}_{\mathrm{u}}) \to \mathcal{P}(\mathbb{D}_{\mathrm{b}} \times \mathbb{D}_{\mathrm{u}})$
- It's trivial, use the obvious function that fits the types.
- Except many analyses depend on and also need to influence the result of the base analysis, so actually we have $tf_{\mathrm{u}} \colon \mathbb{D}_{\mathrm{b}} \times \mathbb{D}_{\mathrm{b}} \times \mathbb{D}_{\mathrm{u}} \to \mathbb{D}_{\mathrm{b}} \times \mathbb{D}_{\mathrm{u}}$
- Composition is still very easy: $tf = \mathrm{map}(\lambda(\mathrm{b}, \mathrm{u}).\ tf_{\mathrm{u}}(\mathrm{b}, tf_{\mathrm{b}}(\mathrm{b}), \mathrm{u}))$.

Branches: how does it work?

- Conditional constant propagation
  - "Constant Propagation with Conditional Branches." (Wegman & Zadeck, 1991)
  - Synergy between constant propagation and dead-code elimination.
- When we reach the last statment, the wrong path is eliminated as dead code!

Example

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

$$\{([\mathtt{change\_y} \mapsto 0], \{\mathtt{y}\}), ([\mathtt{change\_y} \mapsto \overline{\{0\}}], \{\mathtt{save_y}, \mathtt{y}\})\}$$

Multithreaded analysis

- Not interesting! The real challenge is doing this inteprocedurally.
- Multithreaded analysis would only add some technical complexities
  - The formulas would look more complicated.
  - Essentially we collect side-effects, and these have to be merged.
- How to deal with function calls?
  - What is our treatment of functions.
  - How do we do interprocedural path-sensitivity.

Functional approach to interprocedural analysis

- Let $\mathcal{F}$ be the set of all procedures in the programn and N denote nodes in the CFG.
- The set of variables $\mathrm{V} = (\mathcal{F} \cup \mathrm{N}) \times \mathbb{D}$ in the system of constraints are:

  Calls: $\langle \mathrm{f}, \mathrm{d} \rangle$, where $\mathrm{f} \in \mathcal{F}$ is a procedure and $\mathrm{d} \in \mathbb{D}$ is the state in which the function was called. The return state of the function is associated with these variables.

  Nodes: $\langle \mathrm{n}, \mathrm{d} \rangle$, where $\mathrm{n} \in \mathrm{N}$ is a node in the control flow graph of a function f. The second component d denotes the state in which the function f was called.

- The system is infinite, and is solved by demand-driven solvers.
- Example on next slide. . .

Transfer functions for calls

Whenever a function is called, the analyzer uses the following transfer functions to deal with it:

- **entry** computes the function to be called and its entry state.
- **combine** integrates the return value of the call with the local state when the function returns.
- **special** Deals with library functions (it can be seen as part of combine).

Example

```
int f(int y) { return y + 1; }
int main() { x = f(5); }
```

When analyzing the call to f, the analyzer asks for the variable $(\mathtt{f}, [\mathtt{y} \mapsto 5])$ and the resulting state will be $[\mathtt{retvar} \mapsto 6]$, and the combine function will assign it to the variable $\mathtt{x}$.

Interprocedural path-sensitivity

- The job of the base analysis is to resolve function calls, and user analysis only overrides the treatment of library functions.
- Composing these functions simply means:
  - The new entry function only adds the user state to the base entry state.
  - The new combine must consider user provided definitions of library functions.
- We haven't worked out all the details on paper, but it should be easy, although there are small complications
  - Most problems are due to unexpected behaviours of library functions.
  - When the mutex analysis handles a lock function, it should return two states depending on whether the locking succeeds.
  - This is incompatible with our current types of the analysis specifications.

Conclusion

- The analysis transformers lets you transform a very simple specification (40-80 loc) to very sophisticated analysis.
- The next step is to get down to about 10 lines and depend less on O'Caml and specific knowledge of the formalism.
- Since many conditions are very simple ($\forall x : A(x)$ before $B(x)$), the Goblint Analysis Patterns might be good enough.
- Related work!

Thank you!

Thank you for your attention.
Questions and comments are always welcome!