Multi-Stage Construction of a Global Static Analyzer

Basile STARYNKEVITCH

CEA LIST (software reliability lab. Saclay, France)
basile@starynkevitch.net or basile.starynkevitch@cea.fr

july 18, 2007, Ottawa, Gcc Summit

Warning

Work in Progress!

(several KLOC, untested, $\alpha+$, barely integrated)
See `http://starynkevitch.net/Basile/` and
`http://ggcc.info/`

Introduction and Context

global static analysis

- global static analysis - a **niche market** :
  - *costly* computations of
  - *static* properties of a
  - entire *source* program
- uses the **compiler's internal representation[s]** (Gimple/Ssa)
- **whole program** (or whole library) analysis
- **unusual usage** of "mainstream compiler" ($\neq$ code generation)
- existing [commercial] tools show a small **market demand**
  - *critical software* (aerospace, automotive, medical, nuclear)
  - *less critical software* (consumer embedded appliances)
  - for medium sized software (because of whole program analysis)
  - much slower (about $\times 10 - \times 100$) than -O3 compilation
- **too few** libre (free or opensource) software tools for mainstream languages

Introduction and Context

possible applications of global static analysis

1. [for advanced users:] *display contextual properties* of source code ("abstract debugging?")
2. *hazard* (or threat) *detection* (super enhanced contextual "Lint");
   `file.c:345: possible zero-divide in f` or `possible null dereference of p` when called from `otherfile.c:456: with x>3 and x<20`
   challenge = to reduce unavoidable false warnings
3. possible program-wide *optimizations*, e.g.
   - contextual simplification (possibly bypassing tests like non-null, `assert`-s, etc...) and cloning (cf.G. Fursin et al's talk) etc...
   - devirtualization
   - clever cache prefetching
4. *coding rules validation*

Introduction and Context

context: GlobalGCC (ITEA labelled) project

GlobalGCC =

- ITEA labelled (Information Technology for European Advancement)
- a dozen of european (Spanish, French, Italian?) partners (R&D labs, SMEs, industrials) lead by Mandriva
- partly funded by various european public authorities (e.g. MINEFI in France, etc...)
- aims to enhance (& contribute to) GCC with global static analysis
- working on GIMPLE/SSA internal representation
- (so) accepting all of GCC source (front-ends)
- ggcc hosted on Linux (or some other Unixes)

Introduction and Context

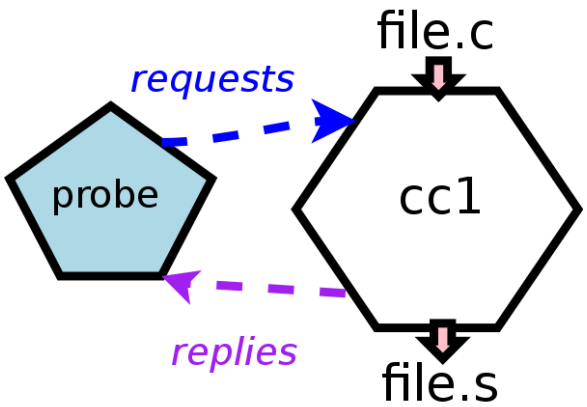
Abstract Interpretation

- Conceptual framework by Patrick and Radhia Cousot (1978)
- already used in GCC (eg CCP)
- idea: abstract (= approximate) on program values (variables, relations, ...) with lattices e.g.
  1. interval of integers (e.g. $\mathtt{i} \in [2; 5]$)
  2. set of linear inequalities (e.g. $3\mathtt{i} + 5\mathtt{j} \leq 22 \wedge 4\mathtt{i} - \mathtt{j} < 6$)
  3. various shapes or graphs for heap data
  4. combination (e.g. cartesian product) of lattices
  5. etc!
- "interpret" the program with the lattices

- "interpret" (with the lattices) the analyzed program
  - [nearly] all the control flow graph, e.g. both branches of tests
  - over-approximation thru widening (e.g. interval $[3; 6]$ becomes $[3; \infty[$) and narrowing
  - so analysis always terminate,perhaps imprecisely (e.g. $\top$ everywhere)
- conceptual result = abstract values at each control point
- profit of additional information (e.g. user conditions on input)
- "interpreter"-like behavior of analyzer (lots of temporary data)
- can be multi-staged: generate specific analyzing code and run it
- difficult to code in C better symbolic processing languages [ML,Lisp]
- existing scalar lattice libraries: Parma Polyhedra Library ♡, Apron ...
- several static analyzers written at CEA LIST (LSL)

Multi-staged infrastructure

Compiler Probe



COMPILER PROBE

- asynchronous $N_{requ} \longrightarrow N_{repl}$ messages
- the probe is an external (e.g. GTK graphical interface) process
- compile- & run- time configurable
- Linux/Unix specific `select`, `SIGIO`, `O_NONBLOCK`
- could be useful to others
- to partly show lots of information
- textual (arbitrary dump compatible) messages
- `cc1` should frequently ($\approx 10Hz$) do `comprobe_check("explain");` macro-expanded to a quick test

Multi-staged infrastructure

Dynamic Run-time

Abstract Interpretation allocates a lot of (complex) *temporary* data, too difficult to manage explicitly.
Current GCC garbage collection (GGC) (marker `GTY` processed by `gengtype`, *explicit* calls to `ggc_collect`) focus on usually permanent data (`tree`-s).
Generational copying GC schemes are better suited to deal with lots of temporary data (some of them cyclic).
Impedance mismatch : *Existing implementations* or HL languages are too different from current GCC datastructures. Many GCC datastructures should be accessd quickly (C macros, no functions).
Basilys runtime: a generational copying GC + a dynamic code loader (micro-plugin) using `libtool` dyn. loader

Multi-staged infrastructure

our copying garbage collector

Basilys copying garbage collector:

- values (discriminated, allocated in a birth region) like e.g.
  1. boxed GCC pointers (`tree`-s, `basic_block`-s, `edge`-s, etc...)
  2. boxed scalars (longs, strings, ...)
  3. Basilys objects (see below)
  4. hash maps of values keyed by objects, or by `tree`-s, etc..
  5. list and tuples of values
  6. closures, routines, ...
  7. finalized pointers PPL types, `FILE`
- minor collection copies live values to GGC heap & frees birth region
- full collection also triggers `ggc_collect`

- Basilys GC implicitly called by allocator (when birth region full)
- local pointers on `cc1` call stack should be known
- "non" intrusive & compatible with GGC and existing GCC passes
- copying schemes rumored to be cache friendly
- write barrier for modification (mini hashtable + store zone at end of birth region)
- inpractical to code for manually in C (e.g. no nested calls)
- easy to generate (C) code for Basilys GC
- no memory overhead (each value starts by its discriminant object)

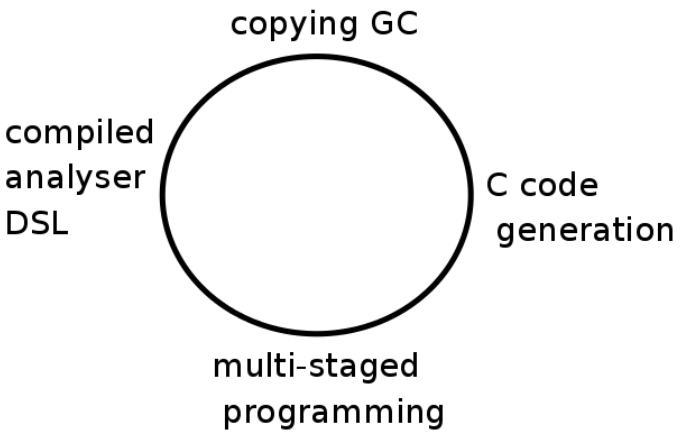
Our objects values have exactly:

1. a class value pointer, their discriminator `obj_class`
2. an unsigned hash code `obj_hash`
3. a short number `obj_num`
4. a short length `obj_len`
5. the pointer to the array of slots `obj_vartab[obj_len]`

Each Basilys value starts with a discriminator pointer, whose `obj_num` determine the C and GGC concrete type.
$\longrightarrow$ Quick GC type machinery, cheap (single-inheritance mono-dispatch) object system (ObjVlisp like: classes, fields, selectors, ... are objects)
Objects used for discriminator (or classes), abstract values or data, ...

compiled dynamic DSL for analyzers

vertuous circle



- complex datastructures required for static analysers
- efficient (copying GC) runtime prefers generated code
- dynamic compiled domain specific language
  - lisp-y like
  - compiled to C
  - tailored to existing GCC data (handle Basilys values & Gcc stuff)
  - enable runtime code generation (thru C & ltdl_dlopenext) for analysis

compiled dynamic DSL for analyzers

language pecularities

- lisp-like syntax (*operator arguments ...*)
- lispy semantics, with few unboxed types (for scalars, GCC `tree`-s & stuff, PPL lattices)
- compiled to simple C (might be Gimple or Llvm?) → meta-programmable
- easy primitives for C stuff:
  `(defprimitive tree2val (:tree u) :value "basilys_boxtree(" u ")")`
- designed to accomodate GCC pecularities (e.g. basic block iterators) with low overhead
- common control structure `defun if lambda forever exit setq let`

- multiple results in functions and in calls
- closure application handles unboxed arguments and multiple results
- small module and macro system; a module compiles into a runtime generated shared object
- extensible for ad-hoc GCC stuff (e.g. `tree` or `basic_block` iterators)
- classes (with method dictionnary), mono-dispatched methods
- quick `is-a` and `instance-of` tests
- runtime introspection (could be a widening strategy) of call stack

compiled dynamic DSL for analyzers

implementation status

- compiler probe implemented with a simple GTK probe
- basilys runtime implemented (including copying GC)
- basilys translator :
  - cold translator (for bootstrapping) implemented in CommonLisp
    1. macro expansion (into specific abstract syntax classes)
    2. normalization e.g. `(f (g x) y)` $\longrightarrow$ `let u = g(x) in f(u,y)`
    3. C-tree generation
    4. C output
  - warm generator incomplete (in Basilys) - should mimic above cold
- analyzers not implemented yet

Lessons

Static analysis passes (and perhaps other features already existing GCC) are complex to code and human productivity requires higher level languages (than C or C++) to implement them

many static analysers (ASTREE, Polyspace?) are coded in ML, but defining a type system for GCC internals is a big work, perhaps impossible

A better runtime can be developped within GGC constraints

A lisp dialect can be designed to be compatible with GCC internals

Future work: clever analysers

- complete the warm Basilys compiler (with ad-hoc GCC iterators)
- develop analysers:
  1. simple scalar analysis
  2. pointer analysis
  3. combining several analysis
  4. introspective techniques for widenning
  5. scalability issues?
- should add persistence (maybe with the help of LTO) to permit larger program analysis.