Case studies: Control-Flow Analysis and Abstract Debugging

Jan Midtgaard

Week 6, Abstract Interpretation

Aarhus University, Q4 - 2012

Last time

A catalogue of abstractions

- Toolbox abstractions
- Structural abstractions: sums, pairs/tuples, ...
- Numerical abstractions: constants, intervals, congruences, polyhedra, ...

Abstract Debugging of Higher-Order Imperative Languages

Basic idea and context

Instead of ’dataflow analysis’ or ’program verification’, an analysis is used for ’abstract debugging’,

i.e. using abstract interpretation to locate the cause of bugs statically (without running the program!)

Achieved through a cool combination of forwards/backwards analysis

Basic idea and context

Instead of ’dataflow analysis’ or ’program verification’, an analysis is used for ’abstract debugging’,

i.e. using abstract interpretation to locate the cause of bugs statically (without running the program!)

Achieved through a cool combination of forwards/backwards analysis

Historic context: *"...applicable to languages such as Pascal, Modula-2, Modula-3, C or C++"*

The paper is from 1993 — Java wasn’t invented until 1995. All examples are given in Pascal

A crash course in Pascal (enough to parse examples)

- imperative programming language
  - types: integers, arrays, ...
  - statement-based: assignment (`:=`), `if`-`then`-`else`, loops (`while`, `for`, `repeat`-`until`)
- lexically scoped, variables are declared with `var`
- blocks are written `begin...end` (instead of `{...}`)
- `read(n)` reads input from stdin and assigns result to variable `n`
- `write(n)` outputs variable `n` to stdout

Pascal peculiarity 1

A function returns its result by "assigning it to the function's name":

```
function Fac(n: integer): integer;
begin
   if n = 0 then
      Fac := 1
   else
      Fac := n * Fac(n-1)
end;
```

Pascal peculiarity 2

Arrays are indexed as indicated by their declaration:

```
program For;
  var i, n : integer;
      T    : array [1..100] of integer;
  begin
      read(n);
      for i := 0 to n do
          read(T[i])
  end.
```

Problem 1: for `i=0` the statement `read(T[i])` indexes the array out-of-bounds

Problem 2: for this program, the input `n` also has to be < 101

Two types of assertions

The debugger is driven by two types of assertions:

Invariant assertions these are similar to normal `assert` statements: properties that must always hold at this point.

Example: `x > 0` at some program point

Intermittent assertions these are different: properties that eventually hold at this point

Properties in collecting semantics

Semantically, these properties can be expressed as a combination of forward/backward/lfp/gfp:

- Descendants of a set of states Σ (forward):

$$\mathrm{lfp}(\lambda X.\, \Sigma \cup post[\tau](X))$$

- Ascendants of a set of states Σ (backward):

$$\mathrm{lfp}(\lambda X.\, \Sigma \cup pre[\tau](X))$$

- Ascendants not leading to error in S_{err} (backward):

$$\mathrm{gfp}(\lambda X.\, pre[\tau](X) \setminus S_{err})$$

Assertion properties, more generally

For a property $\Pi \in \wp(S)$, that will eventually hold:

$$\mathbf{eventually}(\Pi) = \mathrm{lfp}(\lambda X.\, \Pi \cup \mathit{pre}[\tau](X))$$

with the corresponding Kleene sequence:

$$\mathbf{eventually}(\Pi) = \Pi \cup \mathit{pre}[\tau](\Pi) \cup \mathit{pre}^2[\tau](\Pi) \cup \ldots$$

For a property $\Pi \in \wp(S)$, that must always hold:

$$\mathbf{always}(\Pi) = \mathrm{gfp}(\lambda X.\, \Pi \cap \mathit{pre}[\tau](X))$$

with the corresponding Kleene sequence:

$$\mathbf{always}(\Pi) = \Pi \cap \mathit{pre}[\tau](\Pi) \cap \mathit{pre}^2[\tau](\Pi) \cap \ldots$$

Assertions as always/eventually properties

Programs are modeled using $PC \times Memory$ pairs.

Property π_k always holds at point c_k (for all $k \in K_a$)

$$\Pi_a = \{\langle c, m\rangle \in S \mid \forall k \in K_a : c = c_k \implies m \in \pi_k\}$$

(invariant ass.)

Property π_k eventually holds at point c_k (for some $k \in K_e$)

$$\Pi_e = \{\langle c, m\rangle \in S \mid \exists k \in K_e : c = c_k \wedge m \in \pi_k\}$$

(intermittent ass.)

Fixed point computation, (coll.) semantically

Semantically we seek the limit I of the sequence

$$S = I_0 \supseteq I_1 \supseteq I_2 \supseteq I_3 \supseteq \dots$$

where

- $I_{k+1} = \mathrm{lfp}(\lambda X.\, I_k \cap (S_{in} \cup \mathit{post}[\tau](X)))$
- $I_{k+2} = \mathrm{gfp}(\lambda X.\, I_{k+1} \cap \Pi_a \cap \mathit{pre}[\tau](X))$
- $I_{k+3} = \mathrm{lfp}(\lambda X.\, I_{k+2} \cap (\Pi_e \cup \mathit{pre}[\tau](X)))$

The fixed point computation continues to propagate forwards ($k+1$), backwards ($k+2$), backwards ($k+3$)

Error detection from fixed point result

- All $s \in S_{in} \backslash I$ break one of the programmer's invariants, since s is not in Π_a or will not lead to a state in Π_e.

Hence such states can be reported to the programmer.

From fixed point semantics to analysis

The analysis is similar, except it performs fixed point computations over an abstract domain.

The analysis and semantics are (of course) connected by Galois connections.

Minimal use of widening

To speed up convergence or guarantee termination the analysis uses widening/narrowing operators.

Widening (and narrowing) represent information loss, so we want to minimize the number of widenings.

Only loops (cycles) can lead to infinite chains in the analysis.

Convergence is guaranteed by at least one widening operator per cycle in the equation dependency graph.

Interval analysis

The analysis prototype uses an interval lattice that correctly models underflow/overflow:

$$l, u \in [-2^{b-1}; 2^{b-1} - 1]$$

of finite height 2^b. However Bourdoncle still uses widening to speed up convergence.

For strictly increasing upper bounds, interval widening jumps to top ($2^{b-1} - 1$)

and for strictly decreasing lower bounds, interval widening jumps to bottom (-2^{b-1})

Hence the resulting analysis converges in at most 4 iterations

Analysis complexity

One can simply solve the equations by Kleene fixed point iteration.

However there are more clever approaches based on chaotic iteration.

Bourdoncle combines two strategies:

- First compute intraprocedural fixed points, based on the dependency graph,
- then compute interprocedural fixed points, based on the call graph

The resulting algorithm is quadratic in the program size (assuming the number of variables is constant).

Prototype

The prototype implementation consists of

- approx. 20000 lines of C
- incl. 4000 lines of X-window GUI

It first extracts semantic equations, which are subsequently solved.

The prototype is configurable. By default it performs

- a forward analysis,
- two backward analyses, and
- a final forward analysis

McCarthy's 91 function

Bourdoncle analyses (a generalization of) the following benchmark program:

$$MC(n) = \begin{cases} n - 10 & \text{if } \; n > 100 \\ MC(MC(n+11)) & \text{if } \; \leq 100 \end{cases}$$

which is functionally equivalent to:

$$MC(n) = \begin{cases} n - 10 & \text{if } \; n > 100 \\ 91 & \text{if } \; \leq 100 \end{cases}$$

It is interesting for static analysis, because the constant 91 does not appear anywhere in the source text.

McCarthy's 91 function, generalized

Bourdoncle analyses the following generalized benchmark program:

$$MC_k(n) = \begin{cases} n - 10 & \text{if } \; n > 100 \\ {MC_k}^k(n+10k-9) & \text{if } \; \leq 100 \end{cases}$$

which is still functionally equivalent to:

$$MC_k(n) = \begin{cases} n - 10 & \text{if } \; n > 100 \\ 91 & \text{if } \; \leq 100 \end{cases}$$

But now MC_k contains k recursive calls.

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

If we (invariant) assert $\texttt{n} \leq 101$ here,

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

If we (invariant) assert $\mathtt{n} \leq 101$ here,

the analysis proves $\mathtt{m} = 91$ here

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

If we (intermittent) assert $\mathtt{m} = 91$ here,

MC_9 in Pascal

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
                 MC(MC(MC(MC (n + 81)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

the analysis finds that $\texttt{n} \leq 101$ is a necessary condition here

If we (intermittent) assert $\texttt{m} = 91$ here,

MC_9 in Pascal, buggy

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 71)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

MC_9 in Pascal, buggy

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 71)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

If we (intermittent) assert $true$ here,

MC_9 in Pascal, buggy

```
program McCarthy;
  var m, n : integer;

  function MC(n: integer) : integer;
  begin
     if (n > 100) then
        MC := n - 10
     else
        MC := MC(MC(MC(MC(MC(
               MC(MC(MC(MC (n + 71)))))))))
  end;

begin
   read(n);
   m := MC(n);
   writeln(m)
end.
```

the analysis finds that $\texttt{n} \geq 101$ is a necessary termination condition here

If we (intermittent) assert $true$ here,

Syntox tool for download

Bourdoncle keeps a binary executable for download:

`http://web.me.com/fbourdoncle/page18/page6/page6.html`

It is however restricted to

- sparc (Suns),
- solaris (Sun + Solaris) or
- mips (MIPS/Ultrix DECStation)

Let me know if you find a machine (or an emulator) able to run it.

Summary and conclusion

A very nice application of abstract interpretation machinery.

Overall the basic techniques are very well presented.

Hence they are directly applicable to an "abstract 3CM debugger" (which would be a very cool project).

For more complex features (reference parameters with aliasing, recursive function calls, ...) more details are swept under the rug.