Case Studies: Abstract Debugging and Flight Software Verification

Jan Midtgaard

Winter School, Day 5

http://janmidtgaard.dk/aiws15/

Saint Petersburg, Russia, 2015

Yesterday

1. More approximation methods (Cousot-Cousot:JLP92):
   - Relational and attribute independent analysis
   - Inducing, abstracting, approximating fixed points
   - Widening, narrowing
   - Forwards/backwards analysis
2. A catalogue of abstractions
   - Toolbox abstractions
   - Structural abstractions: sums, pairs/tuples, . . .
   - Numerical abstractions: constants, intervals, congruences, polyhedra, . . .
   - Concretization-based abstract interpretation, briefly

   A retrospective on the 3 counter machine analysis

Today

Two papers and then some:

- Extracting data-flow constraints
- Case studies (based on two research articles):
  - *Abstract Debugging of Higher-Order Imperative Languages*, Bourdoncle, PLDI'93
  - *A Static Analyzer for Large Safety-Critical Software*, Blanchet, Cousot, Cousot, Feret, Mauborgne, Miné, Monniaux, and Rival, PLDI'03
- A bit on compositional analysis
- Quickchecking static analysis properties
- Course wrap-up

Constraint Extraction

From fixed points to constraints

Recall: a fixed point of $F^\sharp$ satisfies: $F^\sharp(S^\#) = S^\#$

and a post-fixed point satisfies: $F^\sharp(S^\#) \sqsubseteq S^\#$

Any post-fixed point of $F^\sharp$ is a sound approximation of $\mathrm{lfp}\, F^\sharp$

In our case, $F^\sharp$ is defined as a big (pointwise) join:

$$F^\sharp(S^\#) = X_1 \dot{\sqcup} X_2 \dot{\sqcup} \ldots \dot{\sqcup} X_n \dot{\sqsubseteq} S^\#$$

which is equivalent to:

$$X_1 \dot{\sqsubseteq} S^\# \ \wedge \ X_2 \dot{\sqsubseteq} S^\# \ \wedge \ \ldots \ \wedge \ X_n \dot{\sqsubseteq} S^\#$$

Extracting 3 counter machine constraints (1/5)

```
F#(S#) =  ( <bot,bot,bot>. [1 -> <top, even, even> ] )

  U.
          U.       ( <bot,bot,bot>. [pc+1 -> [var++]#(S#(pc))] )
     P_pc = inc var


  U.
          U.       ( <bot,bot,bot>. [pc+1 -> [var--]#(S#(pc))] )
     P_pc = dec var


  U.
                   ( <bot,bot,bot>. [pc' -> [var==0]#(S#(pc))] )
          U.       U. ( <bot,bot,bot>. [pc'' -> [var<>0]#(S#(pc))] )
P_pc = zero var pc' else pc''


    C. S#
```

Extracting 3 counter machine constraints (2/5)

```
          ( <bot,bot,bot>. [1 -> <top, even, even> ] )  C. S#

  /\
            U.        ( <bot,bot,bot>. [pc+1 -> [var++]#(S#(pc))] ) C. S#
       P_pc = inc var


  /\
            U.        ( <bot,bot,bot>. [pc+1 -> [var--]#(S#(pc))] ) C. S#
       P_pc = dec var


  /\
                      ( <bot,bot,bot>. [pc' -> [var==0]#(S#(pc))] )
            U.       U. ( <bot,bot,bot>. [pc'' -> [var<>0]#(S#(pc))] ) C. S#
P_pc = zero var pc' else pc''
```

Extracting 3 counter machine constraints (3/5)

```
                                  <top, even, even>      C  S#(1)

  /\
          /\                                    [var++]#(S#(pc))    C  S#(pc+1)
     P_pc = inc var

  /\
          /\                                    [var--]#(S#(pc))    C  S#(pc+1)
     P_pc = dec var

  /\
                  ( <bot,bot,bot>. [pc' -> [var==0]#(S#(pc))] )
          /\      U. ( <bot,bot,bot>. [pc'' -> [var<>0]#(S#(pc))] ) C. S#
P_pc = zero var pc' else pc''
```

Extracting 3 counter machine constraints (4/5)

```
                                  <top, even, even>      C  S#(1)

  /\
          /\                                  [var++]#(S#(pc))    C  S#(pc+1)
     P_pc = inc var

  /\
          /\                                  [var--]#(S#(pc))    C  S#(pc+1)
     P_pc = dec var

  /\
                  ( <bot,bot,bot>. [pc' -> [var==0]#(S#(pc))] )     C. S#
          /\      /\ ( <bot,bot,bot>. [pc'' -> [var<>0]#(S#(pc))] ) C. S#
P_pc = zero var pc' else pc''
```

Extracting 3 counter machine constraints (5/5)

```
                                        <top, even, even>      C  S#(1)

  /\
          /\                                  [var++]#(S#(pc))    C  S#(pc+1)
     P_pc = inc var

  /\
          /\                                  [var--]#(S#(pc))    C  S#(pc+1)
     P_pc = dec var

  /\
                                        [var==0]#(S#(pc))       C S#(pc')
          /\                            /\  [var<>0]#(S#(pc))   C S#(pc'')
P_pc = zero var pc' else pc''
```

Extracting 3 counter machine constraints (5/5)

```
                                        <top, even, even>       C   S#(1)

  /\
          /\                                    [var++]#(S#(pc))     C   S#(pc+1)
     P_pc = inc var

  /\
          /\                                    [var--]#(S#(pc))     C   S#(pc+1)
     P_pc = dec var

  /\
                                              [var==0]#(S#(pc))       C S#(pc')
          /\                                  /\  [var<>0]#(S#(pc))   C S#(pc'')
P_pc = zero var pc' else pc''
```

. . . not that far from the data-flow constraints you may have seen earlier.

Constraints or not?

- Analyses based on abstract interpretation and analyses based on constraints are two sides of the same coin.
- Even though not formulated as such, constraints are meant to soundly approximate an underlying (implicit) formal semantics.
- For efficiency, one may want to extract constraints for a formalized analysis and solve them using clever algorithms and data structures (work-list algorithm, chaotic iteration, . . .)

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 — executions will inevitably lead to a situation where such a property holds.

Example: `false` (i.e. bottom) at program exit (meaning end of program not reachable)

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):

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

Assertion properties, more generally

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

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

with the corresponding Kleene sequence:

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

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

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

with the corresponding Kleene sequence:

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

Assertions as always/eventually properties

Programs are modeled using $\wp(PC \times Memory)$

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.)

at all other points c, the memory m is true (anything)

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.)

at all other points c, the memory m is false (non-existing)

Invariant/intermittent ass. are $\mathbf{always}(\Pi_a)$/$\mathbf{eventually}(\Pi_e)$

Fixed point computation, (coll.) semantically

Semantically we seek the limit I of the sequence

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

where

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

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.
- All $s \in post^*[\tau](I) \backslash I$ also break an invariant, since s follows from the forwards flow from I, but not from the backwards flow.

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.

It is expressed as forward and backward "semantic equations".

These equations are similar to the IMP semantics from day 2

(and to the constraints we just extracted).

Forward equation example

$$
\begin{array}{ll}
0: & x_0 = \top \\
1: \texttt{read(i);} & x_1 = [\![\texttt{read(i)}]\!](x_0) \\
2: \texttt{while (i} \leq \texttt{100) do} & x_2 = [\![i \leq 100]\!](x_1) \sqcup [\![i \leq 100]\!](x_3) \\
3: \quad\quad \texttt{i := i + 1} & x_3 = [\![i := i + 1]\!](x_2) \\
4: & x_4 = [\![i > 100]\!](x_1) \sqcup [\![i > 100]\!](x_3)
\end{array}
$$

where

- $[\![-]\!]$ abstract the primitive operations, and
- the x_i's represent an abstract memory per program point

Backward intermittent equation example

$0:$	$x_0 = [\![\texttt{read(i)}]\!]^{-1}(x_1)$
$1:$ `read(i);`	$x_1 = [\![i \leq 100]\!]^{-1}(x_2) \sqcup [\![i > 100]\!]^{-1}(x_4)$
$2:$ `while (i ≤ 100) do`	$x_2 = \alpha(\{10\}) \sqcup [\![i := i + 1]\!]^{-1}(x_3)$
$3:$ `i := i + 1`	$x_3 = [\![i \leq 100]\!]^{-1}(x_2) \sqcup [\![i > 100]\!]^{-1}(x_4)$
$4:$	$x_4 = x_4$

where

- the intermittent assertion `i = 10` has been inserted (to mimic join with Π_e), and
- $[\![-]\!]^{-1}$ abstract the backwards primitive operations.

Backward invariant equation example

$0:$		$x_0 = [\![\texttt{read(i)}]\!]^{-1}(x_1)$
$1:$	`read(i);`	$x_1 = [\![i \leq 100]\!]^{-1}(x_2) \sqcup [\![i > 100]\!]^{-1}(x_4)$
$2:$	`while (i ≤ 100) do`	$x_2 = \alpha(\{0, 1, 2, \dots\}) \sqcap [\![i := i + 1]\!]^{-1}(x_3)$
$3:$	`i := i + 1`	$x_3 = [\![i \leq 100]\!]^{-1}(x_2) \sqcup [\![i > 100]\!]^{-1}(x_4)$
$4:$		$x_4 = x_4$

where

- the invariant assertion $\texttt{i} \geq \texttt{0}$ has been inserted (to mimic meet with Π_a), and
- $[\![-]\!]^{-1}$ abstract the backwards primitive operations.

Minimal use of widening

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

Widening represents 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.

Forward equations with widening

```
                          x0 = ⊤
read(i);                  x1 = [[read(i)]](x0)
while (i ≤ 100) do        x2 = x2 ▽ ([[i ≤ 100]](x1) ⊔ [[i ≤ 100]](x3))
    i := i + 1            x3 = [[i := i + 1]](x2)
                          x4 = [[i > 100]](x1) ⊔ [[i > 100]](x3)
```

where

- the widening operator breaks the x_2–x_3–x_2 dependency cycle of the above equations

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 the extreme upper bound $(2^{b-1} - 1)$

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

Hence the resulting analysis converges in at most 4 iterations (per variable)

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 ($k \geq 1$):

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

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 `n` ≤ 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 $\mathtt{n} \leq 101$ is a necessary condition here

If we (intermittent) assert $\mathtt{m} = 91$ 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 $\mathtt{n} \geq 101$ 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.

A Static Analyzer for Large Safety-Critical Software

[see PLDI'03 slides]

The ASTRÉE design refinement algorithm

1. Run static analysis with false alarms
2. Manually inspect cause of false alarm
3. Possible causes
   - Either rewrite abstract transfer function to strengthen it,
   - Refine a widening which is too coarse, or
   - Design a new abstract domain that can express the missed invariant
4. Wash, rinse, repeat

Compositional semantics and analysis (1/2)

ASTRÉE is based on *compositional* semantics and analysis: the semantics (or analysis) of a compound statement is a combination of the semantics (or analysis) of its parts.

For example:

$$[\![\texttt{skip}]\!]^\sharp(E^\sharp) = E^\sharp$$

$$[\![S_1 \,;\, S_2]\!]^\sharp(E^\sharp) = [\![S_2]\!] \circ [\![S_1]\!]^\sharp(E^\sharp)$$

$$[\![\texttt{if } c \texttt{ then } S_1 \texttt{ else } S_2]\!]^\sharp(E^\sharp) = [\![S_1]\!](\mathit{true}^\sharp(E^\sharp, c)) \sqcup^\sharp [\![S_2]\!](\mathit{false}^\sharp(E^\sharp, c))$$

$$[\![\texttt{while } c \texttt{ do } S]\!]^\sharp(E^\sharp) = \mathrm{lfp}\, F^\sharp$$

where $F^\sharp = \lambda.\Phi.\ \lambda.E^\sharp.\ \mathit{false}^\sharp(E^\sharp, c) \sqcup^\sharp \Phi([\![S]\!](\mathit{true}^\sharp(E^{\sharp\prime}, c)))$

The classical transition system semantics and derived analyses we have studied do not have this property.

Compositional semantics and analysis (2/2)

The classical transition system semantics and derived analyses we have studied do not have this property.

Why? Harder, as the 3CM language is not structured: we allow arbitrary jumps.

A compositional semantics allows us to reason by structural induction (if we can write one).

A corresponding compositional analysis then inherits this structure and avoids a global fixpoint computation:

- The analysis of sequential code is sequential
- The analysis of a loop involves a loop (namely fixpoint iteration)

In the end the resulting analysis is more efficient.

When combined with Bourdoncle's *minimal widening insight* the resulting analysis is also more precise.

Quickchecking Static Analysis Properties

[see other slides]

Summary

Summary

Two papers and then some:

- Extracting data-flow constraints
- Two case studies based on research articles:
  - *Abstract Debugging of Higher-Order Imperative Languages*, Bourdoncle, PLDI'93
  - *A Static Analyzer for Large Safety-Critical Software*, Blanchet, Cousot, Cousot, Feret, Mauborgne, Miné, Monniaux, and Rival, PLDI'03
- A bit on compositional analysis
- Quickchecking static analysis properties
- Course wrap-up

Course retrospective and wrap-up

Recap of promises

[Abstract interpretation] is simply an alternative view — an eye opener to a new world.

It can be used to develop new techniques, to explain existing ones, to extend or strengthen them (e.g, using disjunctive completion, forward/backward analysis, . . .), to formalize them.

[You are now] in a position to make an informed opinion

It is not just an academic theory: it has been used to check/verify flight control software for both Airbus and Mars missions.

It [has been] hairy: there [was] mathematics and semantics

Learning outcomes and competences

On Friday afternoon, the participants should be able to:

- *describe* and *explain* basic analyses in terms of classical abstract interpretation.
- *apply* and *reason* about Galois connections.
- *implement* abstract interpreters on the basis of the derived program analysis.

In some sense: *thinking tools*

Learning outcomes and competences

On Friday afternoon, the participants should be able to:

- *describe* and *explain* basic analyses in terms of classical abstract interpretation.
- *apply* and *reason* about Galois connections.
- *implement* abstract interpreters on the basis of the derived program analysis.

In some sense: *thinking tools*

Suggestions for additions and changes are very welcome!

You know Kung-fu

You know Kung-fu

. . . I'm here for more practice this afternoon. . .