CS 3110

Proofs about Programs

Prof. Clarkson
Fall 2019

Today's music: Theme from *Downton Abbey* by John Lunn

CLICKER QUESTION 1

Review

Previously in 3110:

- Functional programming
- Modular programming
- Efficiency
- Interpreters

Next unit of course: proofs about programs

Today:

- Equational reasoning
- Proving correctness of recursive functions





Approaches to validation [lec 10]

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - Static analysis
    (“lint” tools, FindBugs, …)
  - Fuzzers
- Mathematical
  - Sound type systems
  - “Formal” verification

Less formal: Techniques may miss problems in programs

All of these methods should be used!

Even the most formal can still have holes:

- did you prove the right thing?
- do your assumptions match reality?

More formal: eliminate *with certainty* as many problems as possible.

Verification

- In the 1970s, scaled to about tens of LOC
- Now, research projects scale to real software:
  - CompCert: verified C compiler
  - seL4: verified microkernel OS
  - Ynot: verified DBMS, web services
  - Four color theorem
  - Project Everest: verified HTTPS stack [in progress]
  - Etc.

In another 40 years?

Our goals

- Write pure functional programs
  - no side effects, mutability, I/O; always terminating
  - integers, lists, options, trees
- Prove correctness theorems
  - CS 2800 mathematics: induction, logic

- Non goal: full verification of large programs

CORRECTNESS

Specifications

```
(** [fact n] is [n] factorial,
    i.e., [n!].
    Requires: [n >= 0]. *)
let rec fact n = ...
```

Postcondition gives an equality between function output and an English/mathematical description involving input

Correctness proofs

- Based on equality between expressions

- When does e = e'?
  - Not asking about OCaml Boolean equality
  - Asking whether two pieces of code are equal…

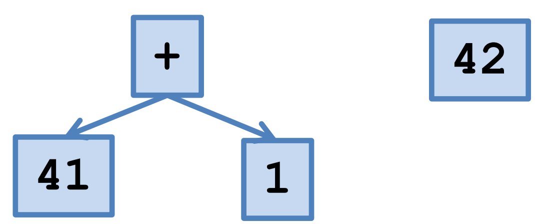
Equality of expressions

$$41 + 1 \stackrel{?}{=} 42$$

Semantically: yes

$$41 + 1 \longrightarrow^{*} 42$$
$$42 \longrightarrow^{*} 42$$

Syntactically: no



Equality of expressions

`fun x -> x` $\stackrel{?}{=}$ `fun y -> y`

Extensionality

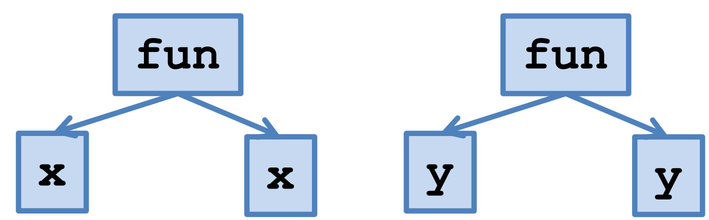
Semantically: yes

for all v,

`(fun x -> x) v` $\longrightarrow^*$ `v`

`(fun y -> y) v` $\longrightarrow^*$ `v`

Syntactically: no



Equality of expressions

$$e = e'$$

if e and e' evaluate to the same value

must be well typed, pure, total

EQUATIONAL REASONING

Example 1

```
let twice f x = f (f x)
let compose f g x = f (g x)
```

twice h x = h (h x) (by evaluation)

compose h h x = h (h x) (by evaluation)

so

twice h x = compose h h x (by transitivity)

Example 1

```
let twice f x = f (f x)
let compose f g x = f (g x)
```

  twice h x
= {by evaluation}
  h (h x)
= {by evaluation}
  compose h h x

Example 2

```
let (<<) = compose
```

Theorem: composition is associative.

(f << g) << h = f << (g << h)

Proof: by extensionality, we need to show:

forall x, ((f << g) << h) x = (f << (g << h)) x

```
let compose f1 f2 x = f1 (f2 x)
```

Example 2

$$((f << g) << h)\ x \quad = \quad (f << (g << h))\ x$$

$((f << g) << h)\ x$

$=$ { evaluation }

$(f << g)\ (h\ x)$

$=$ { evaluation }

$f\ (g\ (h\ x))$

$(f << (g << h))\ x$

$=$ { evaluation }

$f\ ((g << h)\ x)$

$=$ { evaluation }

$f\ (g\ (h\ x))$

QED

PROOFS WITH RECURSION

Summation

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

```
let rec sumto n =
  if n = 0 then 0
  else n + sumto (n - 1)
```

$\texttt{sumto n} \stackrel{?}{=} \texttt{n * (n + 1) / 2}$

Induction on natural numbers

Theorem: forall natural numbers n, P(n).

Proof: by induction on n

Base case: $n = 0$
Show: $P(0)$

Inductive case: $n = k+1$
IH: $P(k)$
Show: $P(k+1)$

QED

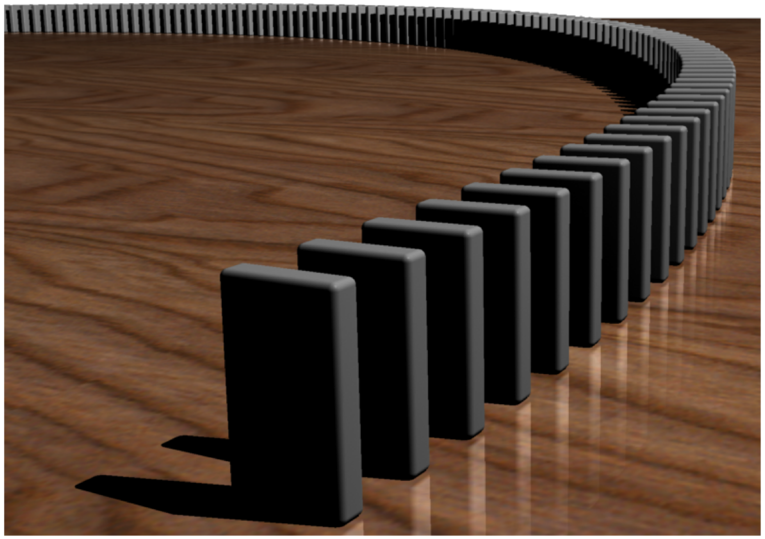
Induction principle

forall properties P of natural numbers,

  if P 0

  and (forall n, P n implies P (n+1))

  then (forall n, P n)



Summation: proof structure

Claim: forall n, sumto n = n * (n + 1) / 2

Proof: by induction on n.
P(n) = sumto n = n * (n + 1) / 2

Base case: n = 0
Show: sumto 0 = 0 * (0 + 1) / 2

Inductive case: n = k + 1
IH: sumto k = k * (k + 1) / 2
Show: sumto (k + 1) = (k + 1) * ((k + 1) + 1) / 2

Summation: base case

Base case: n = 0

Show: sumto 0 = 0 * (n + 1) / 2

  sumto 0

= { evaluation }

  0

= { algebra (or evaluation) }

  0 * (0 + 1) / 2

```
let rec sumto n =
  if n = 0 then 0
  else n + sumto (n - 1)
```

Summation: inductive case

Inductive case: n = k + 1
IH: sumto k = k * (k + 1) / 2
Show: sumto (k + 1) = (k + 1) * ((k + 1) + 1) / 2

  sumto (k + 1)
= { evaluation }
  k + 1 + sumto k
= { IH }
  k + 1 + k * (k + 1) / 2
= { algebra }
  (k + 1) * ((k + 1) + 1) / 2

QED

```
let rec sumto n =
  if n = 0 then 0
  else n + sumto (n - 1)
```

Factorial

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)
```

“i” suggests iterative

```
let rec facti acc n =
  if n = 0 then acc
  else facti (acc * n) (n - 1)
```

```
let fact_tr n = facti 1 n
```

Factorial: correctness

Claim: forall n, fact n = facti 1 n

Proof: by induction on n.
P(n) = fact n = facti 1 n

Base case: n = 0
Show: fact 0 = facti 1 0

Inductive case: n = k + 1
IH: fact k = facti 1 k
Show: fact (k + 1) = facti 1 (k + 1)

Factorial: inductive case

Inductive case: n = k + 1

IH: fact k = facti 1 k

Show: fact (k + 1) = facti 1 (k + 1)

fact (k + 1)	facti 1 (k + 1)
= { evaluation }	= { evaluation }
(k + 1) * fact k	facti (k + 1) k
= { IH }	
(k + 1) * facti 1 k	

STUCK

want to move (k + 1) into accumulator:

facti (k + 1) k

but how?

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)

let rec facti acc n =
  if n = 0 then acc
  else facti (acc * n) (n - 1)
```

Strengthened IH

What we have from IH:

(k + 1) * fact k = (k + 1) * facti 1 k

What we want:

(k + 1) * fact k = facti (k + 1) k

can multiply (k + 1) into acc

So strengthen P(n) to give us what we want:

forall p, p * fact n = facti p n

can multiply anything into acc

Factorial: strengthened ind. case

Inductive case: n = k + 1

IH: forall p, p * fact k = facti p k

Show: forall p, p * fact (k + 1) = facti p (k + 1)

p * fact (k + 1)

= { evaluation }

(p * (k + 1)) * fact k

= { IH with p := p * (k + 1) }

facti (p * (k + 1)) k

facti p (k + 1)

= { evaluation }

facti (p * (k + 1)) k

QED

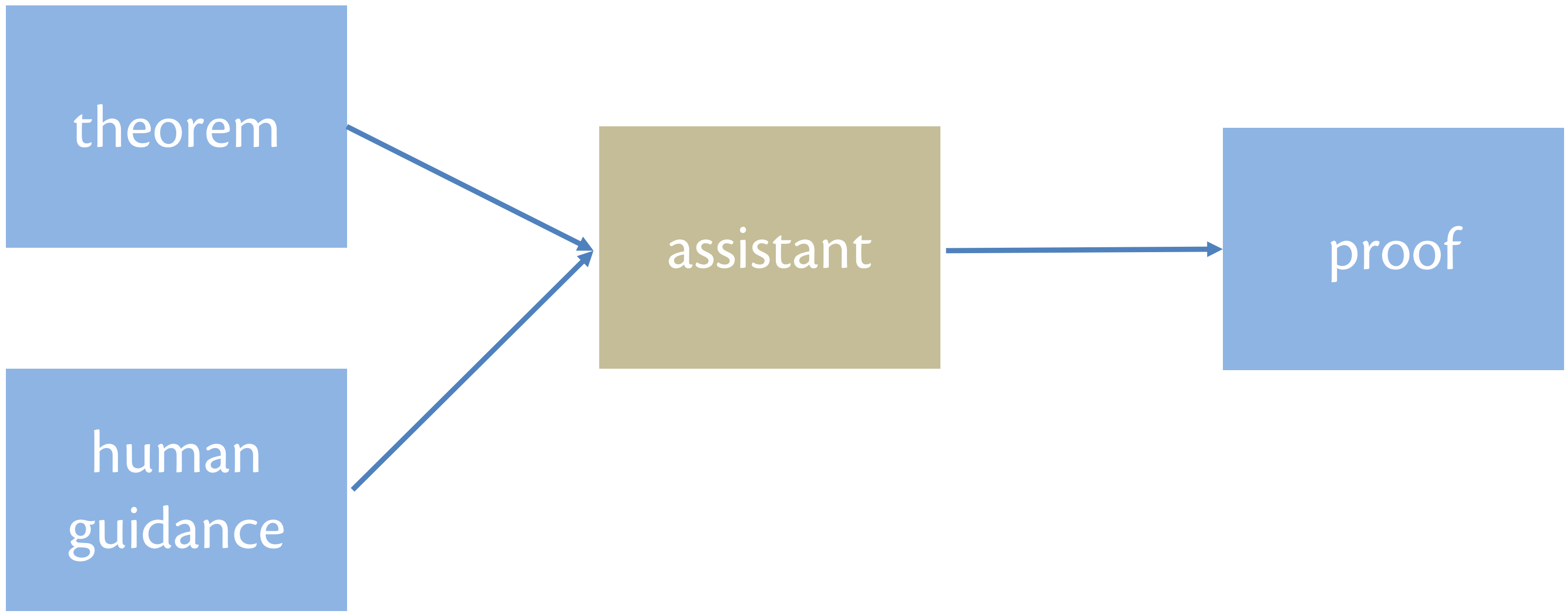
```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)

let rec facti acc n =
  if n = 0 then acc
  else facti (acc * n) (n - 1)
```

Sponsored Advertisement

CS 4160: FORMAL VERIFICATION

Proof assistant

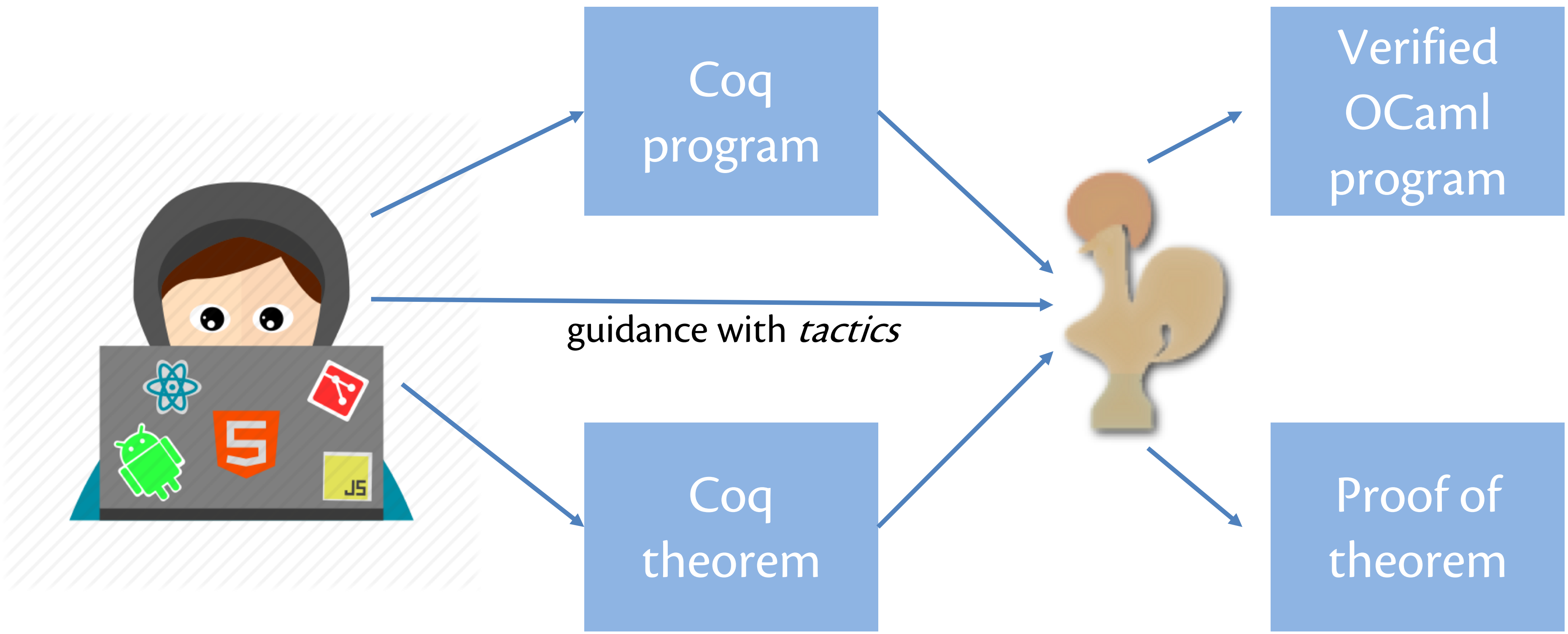


Coq

- **1984:** Coquand and Huet implement Coq based on *calculus of inductive constructions*
- **1992:** Coq ported to Caml
- Now implemented in OCaml

Thierry Coquand
1961 –

Coq for program verification



Demo

CAUTION: HIGHLY ADDICTIVE

Upcoming events

- [Wed] A6 due

This is formal.

THIS IS 3110