FUNCTIONS ARE PROOFS!
AN INTRODUCTION TO THE F* LANGUAGE

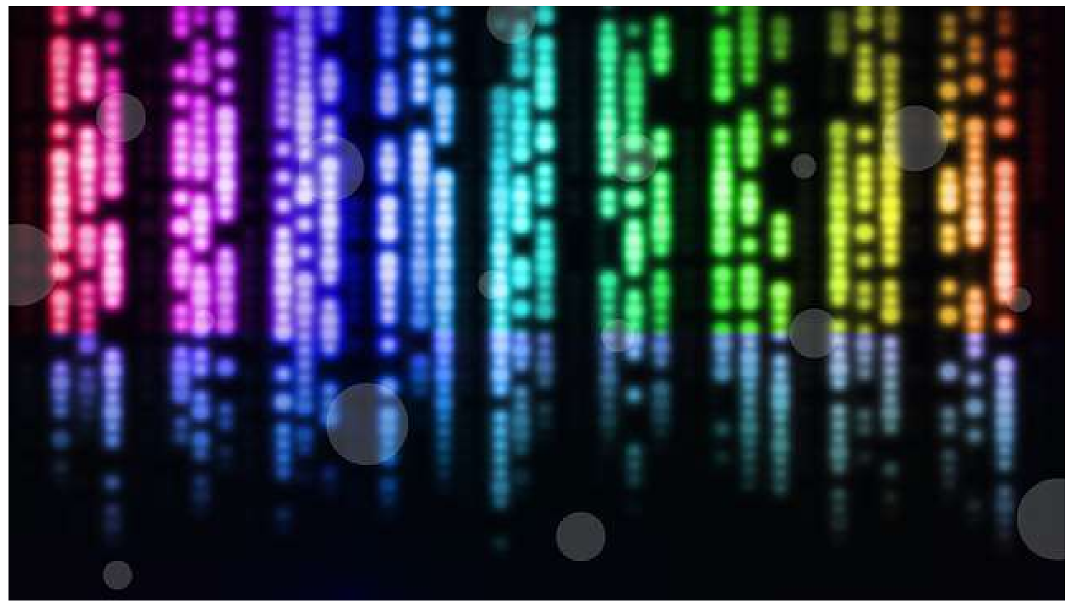
Mark Gritter
@markgritter
MinneBar 2022

SOFTWARE (STILL) SUCKS

Software is hard.

"Try very hard and think a lot" is a failed solution.

Even very smart people miss simple edge cases.



GOAL: HAVE THE COMPUTER CHECK MY BULLSHIT

There are lots of things I *think* are true about my programs.

What if I could tell the computer about them? And have it check if I'm correct?

(And, maybe, use them for optimization?)

Photo by No Revisions on Unsplash

F*: A PROOF-ORIENTED PROGRAMMING LANGUAGE

From Microsoft Research:

- "F* (pronounced F star) is a general-purpose functional programming language with effects aimed at program verification."
- Sort of like a proof assistant, backed by an SMT solver.
- But compiles to real code! (OCaml, F#, C, WASM, assembly)
- Active work: verified implementations of TLS, cryptographic primitives, parsers

Home page: https://www.fstar-lang.org/

Source (+ wiki): https://github.com/FStarLang/FStar/

Online editor: https://www.fstar-lang.org/run.php

HELLO, WORLD

F* is a lot like OCaml.

- If you're unsure what the syntax is, try OCaml first

```
let rec factorial (n:nat) : nat
= if n = 0 then 1
  else op_Multiply n (factorial (n - 1))
```

REFINEMENT TYPES

You may be familiar with statically-typed programming languages.

- Basic types: string, int, bool, etc.
- Checked by the compiler
- Extend with typedefs, composite types, decorators, etc.

F* also has **Refinement types:** an existing type can be augmented with an expression that must be true.

REFINEMENT TYPES: EXAMPLES

A natural number is an integer greater than 0:

```
type nat = n:int{n > 0}
```

A nonempty list:

```
l:(list int){length l > 0}
l:(list int){Cons? l}
```

A square number:

```
n:int{exists (a:int). op_Multiply a a = n}
```

TYPE-CHECKING IS MODEL CHECKING

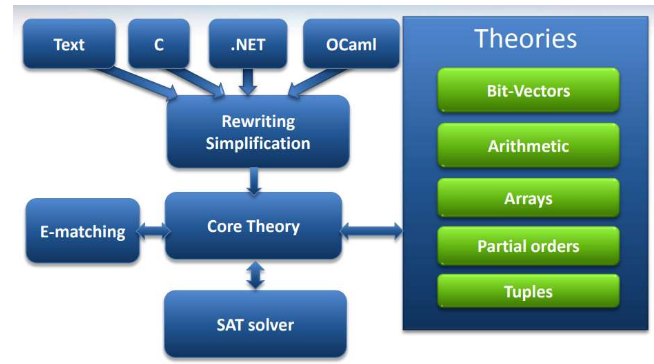
How does F* type-check a function?

```
type odd = n:int{n % 2 = 1}
type even = n:int{n % 2 = 0}
let odd_plus_odd (a:odd b:odd) : even = a + b
let even_plus_odd (a:odd b:even) : even = a + b
```

TYPE-CHECKING IS MODEL CHECKING

F*
type checker

Z3
theorem prover



Does type X match type Y?

Is there a counterexample to statement Z?

```
(assert (not (forall ((@x0 Term) (@x1 Term))
 (implies (and (HasType @x0 EvenOdd.odd)
               (HasType @x1 EvenOdd.even))
          (or label_3
               (= (Prims.op_Modulus (Prims.op_Addition @x0 @x1)
                                    (BoxInt 2))
                      (BoxInt 0))
          )
 )))
```

TYPE-CHECKING IS... PROOF-CHECKING?

We need a couple more pieces:

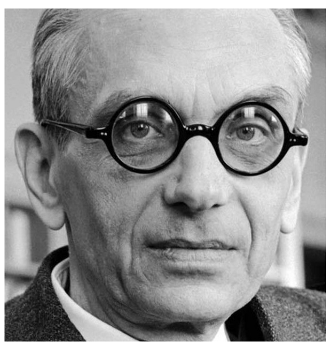
1. Dependent types
2. The horrible pun also known as the Curry-Howard correspondence

THE FOUNDATIONAL PUN (HIGHLY FICTIONALIZED)

LOGIC: The Hypothetical Syllogism

If A implies B, and B implies C, then A implies C!

$$(A \to B) \to ((B \to C) \to (A \to B)$$

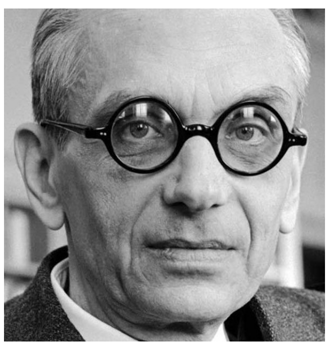


THE FOUNDATIONAL PUN (HIGHLY FICTIONALIZED)

LOGIC: The Hypothetical Syllogism

If A implies B, and B implies C, then A implies C!

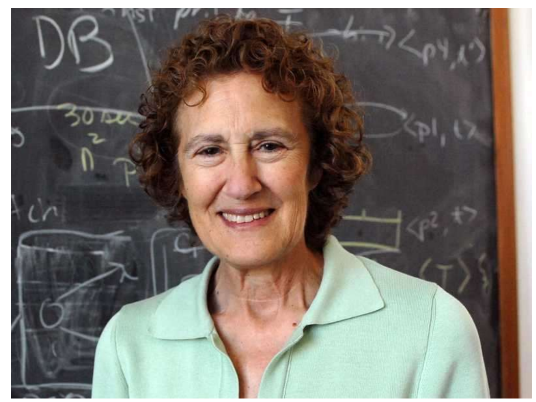
(A→ B) → ((B→ C) → (A→ B)



COMPUTING: Composition

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

(A→ B) → ((B→ C) → (A→ B)



THE FOUNDATIONAL PUN (HIGHLY FIC

LO

If A

THIS IS THE HARDEST PART
(EXCEPT FOR ALL THE OTHER HARD PARTS)

A function is a constructive proof.

Most functions don't prove very interesting theorems.

```
let square (x:int) : int = op_Multiply x x
```

has type

$\text{int} \rightarrow \text{int}$

or, in English, it says

If an integer exists, then an integer exists.

though the form that's maybe more useful is:

For all integers X, some integer exists.

HOW DO WE WRITE A MORE INTERESTING PROOF?

By expanding the language of *types* we can talk about!

Suppose we want to prove

For any prime X, there is some prime Y, and Y is strictly larger than X.

The corresponding theorem in the language of F* is:

```
val larger_prime : (x:prime) -> (y:prime{y>x})
```

The refinement now refers to a previous argument; the jargon for this is dependent typing.

DEPENDENT TYPE EXAMPLES

From the F* standard library:

The **index** function only takes an index that is provably less than the length of the list!

```
val index: #a:Type -> l:list a
   -> i:nat{i < length l} -> Tot a
```

The **filter** function returns a list; every member of that list satisfies the predicate given.

```
val filter : #a: Type -> f:(a -> Tot bool)
   -> l: list a
   -> Tot (m:list a{forall x. memP x m ==> f x})
```

DEPENDENT TYPES: COMPLICATED EXAMPLE

From Advent Of Code 2021, day 21

```
type game_state = {
  player_1_score : (n:nat{n<max_score});
  player_1_position : board_position;
  player_2_score : (n:nat{n<max_score});
  player_2_position : board_position;
  turn : (n:nat{n = 1 \/ n = 2});
  // deterministic die cycles
  // between 1 and 100
  next_die : die_value;
  num_die_rolls: nat
}
```

```
let player_1_move (g:game_state{g.turn=1 /\
g.player_1_score<1000}) :
   (h:game_state{h.turn = 2 /\
    h.num_die_rolls = g.num_die_rolls + 3 /\
    h.player_2_score = g.player_2_score /\
    h.player_2_position = g.player_2_position}) =
  let d0 = g.next_die in
  let d1 = advance_die d0 in
  let d2 = advance_die d1 in
  let next_d = advance_die d2 in
  let new_p = advance_position
g.player_1_position d0 d1 d2 in
  let new_score = g.player_1_score + new_p in
  { g with player_1_score=new_score;
    player_1_position=new_p;
    turn=2;
    next_die=next_d;
    num_die_rolls = 3 + g.num_die_rolls }
```

SUGAR FOR PROOFS

Sometimes, we only care about the proposition, not the value. Or, we may not be able to fit the whole (checkable) proof in a return value.

F* has the type `unit` which has only a single element, written `()`.

You can still refine `unit` with a dependent type, for example:

```
unit{a + b = b + a}
```

So, we can write the commutative property of addition as

```
val commutes (a:int) -> (b:int) -> unit{a + b = b + a}
```

SUGAR FOR PROOFS

```
val commutes (a:int) -> (b:int) -> unit{a + b = b + a}
```

The idiomatic way to write this is

```
val commutes (a:int) (b:int) : Lemma (a + b = b + a)
```

Or, to prove it, we just have to ask the SMT solver for help; just return unit (because that's the only value of the return type!)

```
let commutes (a:int) (b:int) : Lemma (a + b = b + a)
  = ()

let commutes_alt (a:int) (b:int) : Lemma
   (requires True)
   (ensures a + b = b + a) = ()
```

INDUCTIVE PROOFS

Because proofs are "just" functions, we can do all the normal things a functional language provides:

- Take them as arguments
- Call them when we need their results
- Build higher-order proofs
- Recursively have the proof call itself (aka, induction!)

INDUCTION

What's you favorite inductive proof? (Audience participation here!)

Here's an example from the standard library:

```
(** [for_all f l] returns [true] if, and only if, for all elements [x]
appearing in [l], [f x] holds.
let rec for_all_mem
  (#a: Type)
  (f: (a -> Tot bool))
  (l: list a)
: Lemma
  (for_all f l <==> (forall x . memP x l ==> f x))
= match l with
  | [] -> ()
  | _ :: q -> for_all_mem f q
```

BUT THAT'S NOT ALL!

F* also checks the *behavior* of a function, its "computation type", which may include pre- and post-conditions

- `Tot` functions must always terminate and are also `Pure`, no side effects
- `GTot` (ghost total) functions are total functions that do not compute anything; these are used for `Lemma`
- `Dv` (divergent) functions are total functions have no side effect
- `ML` functions may do I/O (and also fail to terminate)

Sometimes you have to give a hint to help F* prove that your function terminates; the tutorial covers this.

MORE

Difference between Boolean operators and Boolean predicates, = vs == vs ===

Reasoning about imperative programs (the **State** effect)

Universes

Custom effects and Dijkstra monads

Tactics and Metaprogramming: automating proofs and generating code

Low*: a subset of F* (plus libraries) that compiles to C

Steel: a language for concurrent programming built on F*

WHY SHOULD YOU CARE?

You might be a PL geek.

You could benefit from learning about a different approach.

- "Proving a program, or even just writing down a specification for it, forces you to think about aspects of your program that you may never have considered before."

You want to prove something?

- Other proof assistants might be more friendly and mature.

You need to write high-assurance software…

- Modelling languages like TLA+ or Alloy are probably easier to get started with (but don't generate code!)
- F* is very much a research language.
- Maybe investigate Dafny and Liquid Haskell

F* RESOURCES

New tutorial under construction: http://www.fstar-lang.org/tutorial/

Hidden old tutorial which is more complete but less interactive: https://fstar-lang.org/tutorial/tutorial.html

Tactics tutorial: https://people.csail.mit.edu/cpitcla/fstar.js/TacticsTutorial.html

Project Everest Slack (invite-only?)

My Advent-of-Code in F* repository: https://github.com/mgritter/aoc-fstar

- I wrote some notes on features missing from the tutorial: https://github.com/mgritter/aoc-fstar/blob/main/doc/FeaturesForTheCompleteNoob.md
- Video series on YouTube: https://www.youtube.com/playlist?list=PLVoZsDupSnwHtYDM4VCMG-HD4CGHOoHOP

The Little Typer! (for an introduction to dependent types)

ABOUT ME

Currently at Akita Software, working on the completely opposite approach: bottom-up, no-code inference of API behavior.

Previously
- Vault team at HashiCorp,
- co-founder Tintri

Get in touch
- Twitter: `markgritter`
- Email: `mgritter@gmail.com`
- GitHub: `mgritter`

PRACTICAL F* TIPS AND TRICKS

When stuck:

- Type ascription operator `<:`
- `assert()` a lot
- `assume()` what you need and come back later (if it works)
- Dump out everything F* knows about the current types with one weird trick:
  - `assert_by_tactic True (fun () -> dump "here")`
- Check you haven't swapped `/\` and `\/`.
- Give S3 more time with –rlimit (sometimes works)
- Let F* unroll recursive definitions more with –fuel (hardly ever works)