Make Illegal States Unrepresentable

“KC” Sivaramakrishnan

IIT MADRAS

Goal of this lecture

Goal of this lecture

- Develop an intuition to frame *invariants* about programs

Goal of this lecture

- Develop an intuition to frame *invariants* about programs
- Use *F* programming language* to express those invariants
  - ✦ F* compiler *statically* (at compile time) enforces these invariants
  - ✦ Avoid checking invariants at runtime

Goal of this lecture

- Develop an intuition to frame *invariants* about programs
- Use *F* programming language* to express those invariants
  - ✦ F* compiler *statically* (at compile time) enforces these invariants
  - ✦ Avoid checking invariants at runtime
- Learn basic concepts about functional programming and refinement types

Goal of this lecture

- Develop an intuition to frame *invariants* about programs
- Use *F* programming language* to express those invariants
  - ✦ F* compiler *statically* (at compile time) enforces these invariants
  - ✦ Avoid checking invariants at runtime
- Learn basic concepts about functional programming and refinement types
- Running example — interpreter for arithmetic & conditional expressions

Goal of this lecture

- Develop an intuition to frame *invariants* about programs
- Use *F* programming language* to express those invariants
  - ✦ F* compiler *statically* (at compile time) enforces these invariants
  - ✦ Avoid checking invariants at runtime
- Learn basic concepts about functional programming and refinement types
- Running example — interpreter for arithmetic & conditional expressions

Questions welcome throughout the talk!

Our language

```
<expr> := ℤ
        | <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>
        | <expr> / <expr>
        | <expr> == <expr>
        | if <expr> then <expr> else <expr>
```

Our language

```
<expr> := ℤ
        | <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>
        | <expr> / <expr>
        | <expr> == <expr>
        | if <expr> then <expr> else <expr>
```

For example

```
if 5 + 6 == 11 then
  12 / 6
else 3 * 4
```

is a valid expression

Interpreter in C

```
<expr> := ℤ
        | <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>
        | <expr> / <expr>
        | <expr> == <expr>
        | if <expr> then <expr>
          else <expr>
```

```c
typedef enum {
  Const, Add, Sub, Mul, Div, Eq, Ite
} expr_tag;

typedef struct expr {
  expr_tag tag;
  union {
    int int_value;
    struct expr** args;
  } info;
} expr;
```

Interpreter in C

```
<expr> := ℤ
        | <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>
        | <expr> / <expr>
        | <expr> == <expr>
        | if <expr> then <expr>
          else <expr>
```

```
if 5 + 6 == 11 then
  12 / 6
else 3 * 4
```

```c
typedef enum {
  Const, Add, Sub, Mul, Div, Eq, Ite
} expr_tag;

typedef struct expr {
  expr_tag tag;
  union {
    int int_value;
    struct expr** args;
  } info;
} expr;
```

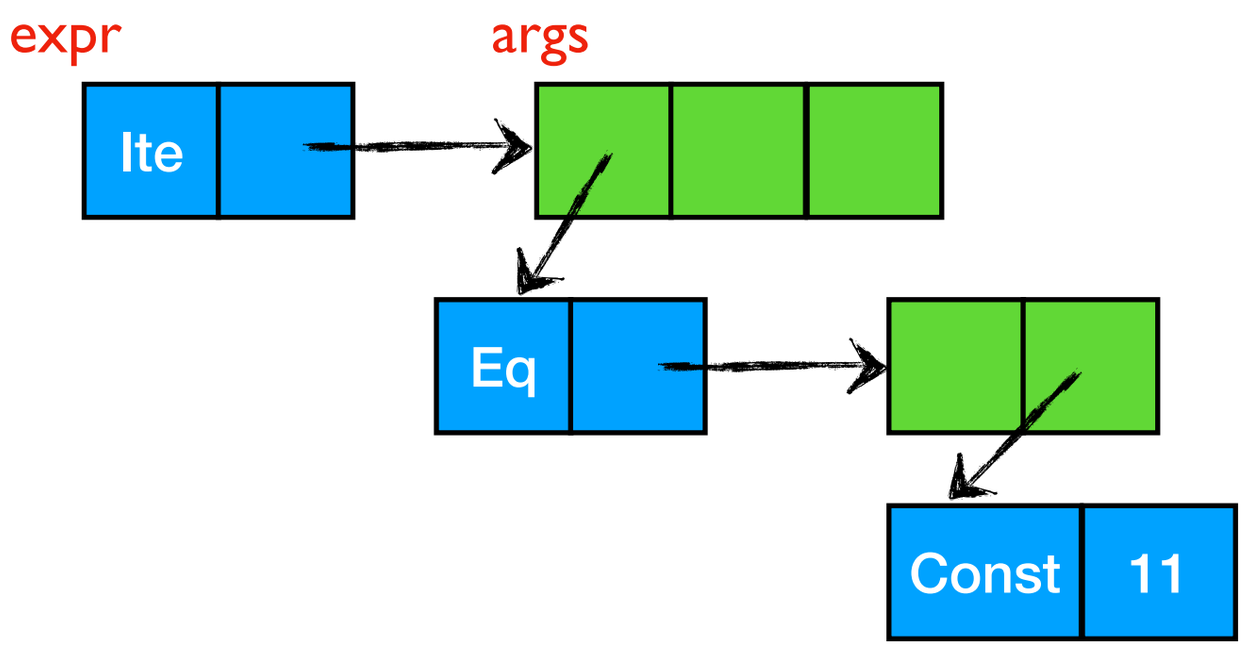
Interpreter in C

```
<expr> := ℤ
        | <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>
        | <expr> / <expr>
        | <expr> == <expr>
        | if <expr> then <expr>
          else <expr>
```

```
if 5 + 6 == 11 then
  12 / 6
else 3 * 4
```

```
typedef enum {
  Const, Add, Sub, Mul, Div, Eq, Ite
} expr_tag;

typedef struct expr {
  expr_tag tag;
  union {
    int int_value;
    struct expr** args;
  } info;
} expr;
```



Demo: interp.c

Invariants

- Although the code works for the example, there are several invariants not handled by the code
  - ✦ An expression may be NULL
  - ✦ The argument array of an expression may be NULL
  - ✦ An expression may not have the right number of arguments
  - ✦ The denominator of the division may be 0

Invariants

- Others invariants are less clear
  - ✦ The first argument of the if-then-else expression may not be a predicate (==)
    - ✤ Should `if 5 + 6 then 24 else 23` be allowed?
  - ✦ Integer overflow not handled
  - ✦ Memory safety not ensured
    - ✤ corrupted, non-NULL pointers for expressions
    - ✤ A not-NULL expression may not always be a well-formed expression

F* programming languages

F* programming languages

- A solver-aided (Z3) general purpose programming language
  - ✦ Functional programming language
  - ✦ Advanced type system to express rich invariants

F* programming languages

- A solver-aided (Z3) general purpose programming language
  - ✦ Functional programming language
  - ✦ Advanced type system to express rich invariants
- Programs can be extracted to OCaml, F#, C, Wasm and ASM.

F* programming languages

- A solver-aided (Z3) general purpose programming language
  - ✦ Functional programming language
  - ✦ Advanced type system to express rich invariants
- Programs can be extracted to OCaml, F#, C, Wasm and ASM.
- Main use case is Project Everest at Microsoft — a drop in replacement for HTTPS stack
  - ✦ Verified implementations of TLS 1.2 and 1.3, and underlying cryptographic primitives
  - ✦ Also in, Mozilla Firefox, WireGuard etc.

Our language — v1

```
<expr> := ℤ
        | <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>
```

Demo: interpv1.fst

Invariants

- ~~An expression may be NULL~~
- ~~The argument array of an expression may be NULL~~
  - *No NULL pointers in F*!*
- ~~An expression may not have the right number of arguments~~
  - *Constructors take arguments!*
- The denominator of the division may be 0
- The first argument of the if-then-else expression may not be a predicate (==)
- ~~Integer overflow not handled~~
  - *Infinite precision integers by default*
- ~~Memory safety not ensured~~
  - *Garbage collection*

Our language — v2

```
<expr> := ℤ
        | <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>
        | <expr> / <expr>
```

Demo: interpv2.fst

Invariants

- ~~An expression may be NULL~~
- ~~The argument array of an expression may be NULL~~
- ~~An expression may not have the right number of arguments~~
- ~~The denominator of the division may be 0~~
  - *F* enforces denominator to be non-zero (subtyping)*
- The first argument of the if-then-else expression may not be a predicate (==)
- ~~Integer overflow not handled~~
- ~~Memory safety not ensured~~

Our language — v3

```
<expr> := ℕ
        | <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>
        | <expr> / <expr>
```

Demo: interpv3.fst

Our language — v4

```
<const> := ℕ | Bool
<expr>  := <const>
         | <expr> + <expr>
         | <expr> - <expr>
         | <expr> * <expr>
         | <expr> / <expr>
         | <expr> == <expr>
         | if <expr> then <expr> else <expr>
```

Our language — v4

```
<const> := ℕ | Bool
<expr>  := <const>
         | <expr> + <expr>
         | <expr> - <expr>
         | <expr> * <expr>
         | <expr> / <expr>
         | <expr> == <expr>
         | if <expr> then <expr> else <expr>
```

Must reject

```
if 5 + 6 then 23 else 24
```

Our language — v4

```
<const> := ℕ | Bool
<expr>  := <const>
         | <expr> + <expr>
         | <expr> - <expr>
         | <expr> * <expr>
         | <expr> / <expr>
         | <expr> == <expr>
         | if <expr> then <expr> else <expr>
```

Must reject

```
if 5 + 6 then 23 else 24
```

Allow only boolean expressions here

Our language — v4

```
<const> := ℕ | Bool
<expr>  := <const>
         | <expr> + <expr>
         | <expr> - <expr>
         | <expr> * <expr>
         | <expr> / <expr>
         | <expr> == <expr>
         | if <expr> then <expr> else <expr>
```

Must reject

```
if 5 + 6 then 23 else 24
```

Allow only boolean expressions here

Add static type checking to our language!

Our language — v4

```
<const> := ℕ | Bool
<expr>  := <const>
         | <expr> + <expr>
         | <expr> - <expr>
         | <expr> * <expr>
         | <expr> / <expr>
         | <expr> == <expr>
         | if <expr> then <expr> else <expr>
```

Must reject

```
if 5 + 6 then 23 else 24
```

Allow only boolean expressions here

Add static type checking to our language!

Demo: interpv4.fst

Invariants

- ~~An expression may be NULL~~
- ~~The argument array of an expression may be NULL~~
- ~~An expression may not have the right number of arguments~~
- ~~The denominator of the division may be 0~~
- ~~The first argument of the if-then-else expression may not be a predicate (==)~~
  - *Encode static types using type-level functions*
- ~~Integer overflow not handled~~
- ~~Memory safety not ensured~~

Summary

- Functional programming offers a concise way to write expressive programs
- Rich type systems help express and enforce program invariants
- Slides + programs available at

https://github.com/prismlab/aicte-compilers-lecture-2021