CS 320: Concepts of Programming Languages

Lecture 10: Joys and Pains of My Life

Ankush Das

Nathan Mull

Oct 10, 2024

Announcements

- Midterm coming up on Oct 17
- Assignment 5 due tonight at 11:59pm
- Next assignment will be released *after* the midterm
- *No lecture on Tue, Oct 15*
- Please study for the midterm! Ask questions on Piazza; come to office hours; discuss with friends, etc.
- This lecture will not be part of the midterm syllabus

Today's Lecture

- *This is my last lecture! Nathan will take over after the midterm!*
- Today, we have a sit back and relax lecture! Take a break from midterm prep
- Now that you're halfway to becoming mathematicians, I can finally explain what makes a PL good or bad
- We will learn the math behind it
- *Disclaimer: this lecture has a lot of personal opinions!*

Need for Formal Semantics

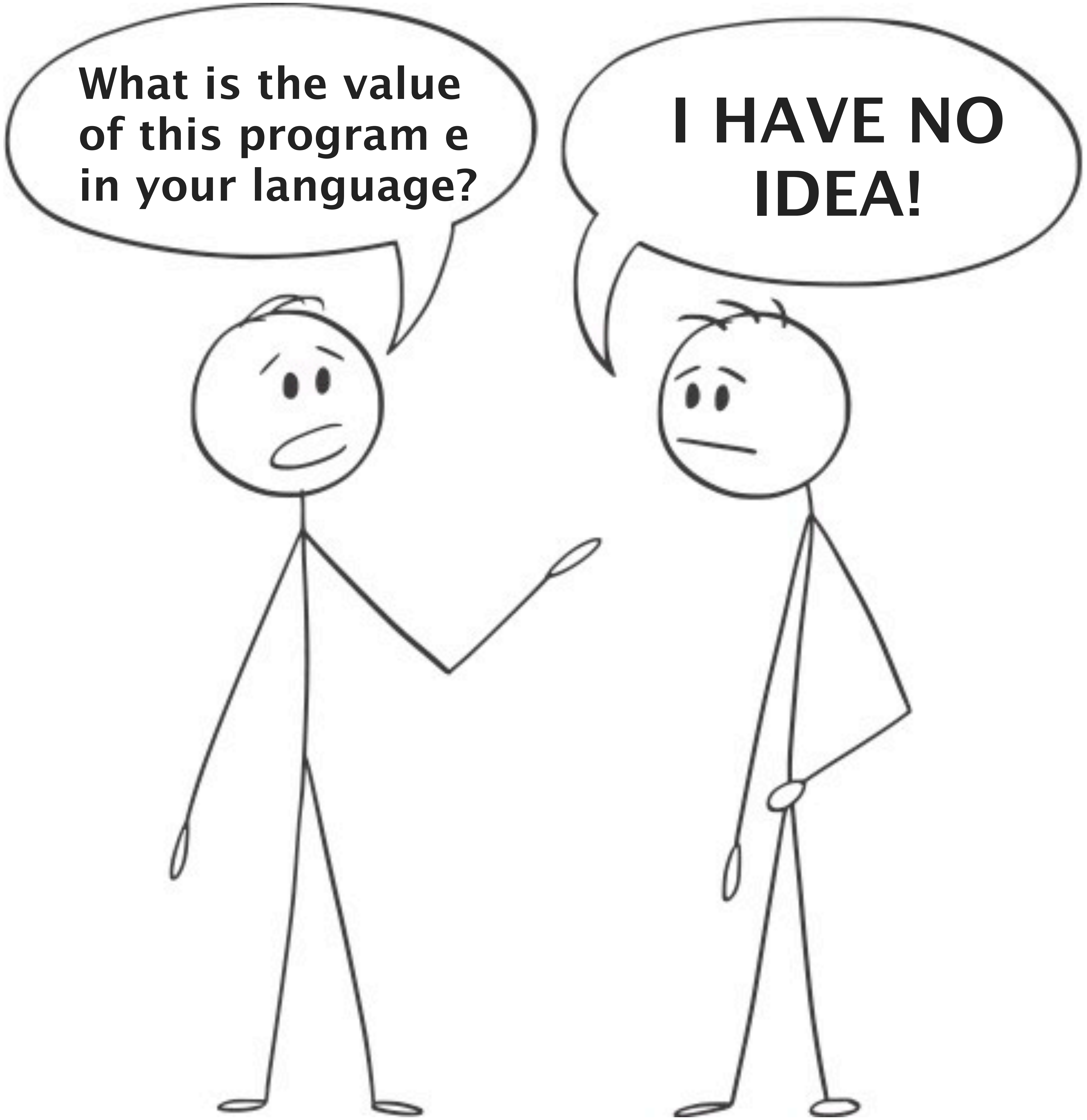
- *First Opinion:*
  A good PL must come with a formal syntax, type system, and semantics!

- Let's see why. Suppose you design a new language and you show it to your friends, or better yet, you release it publicly!

- And your language does not have a formal semantics

Need for Formal Semantics

- *First Opinion:*
  A good PL must come with a formal syntax, type system, and semantics!
- Let's see why. Suppose you design a new language and you show it to your friends, or better yet, you release it publicly!
- And your language does not have a formal semantics

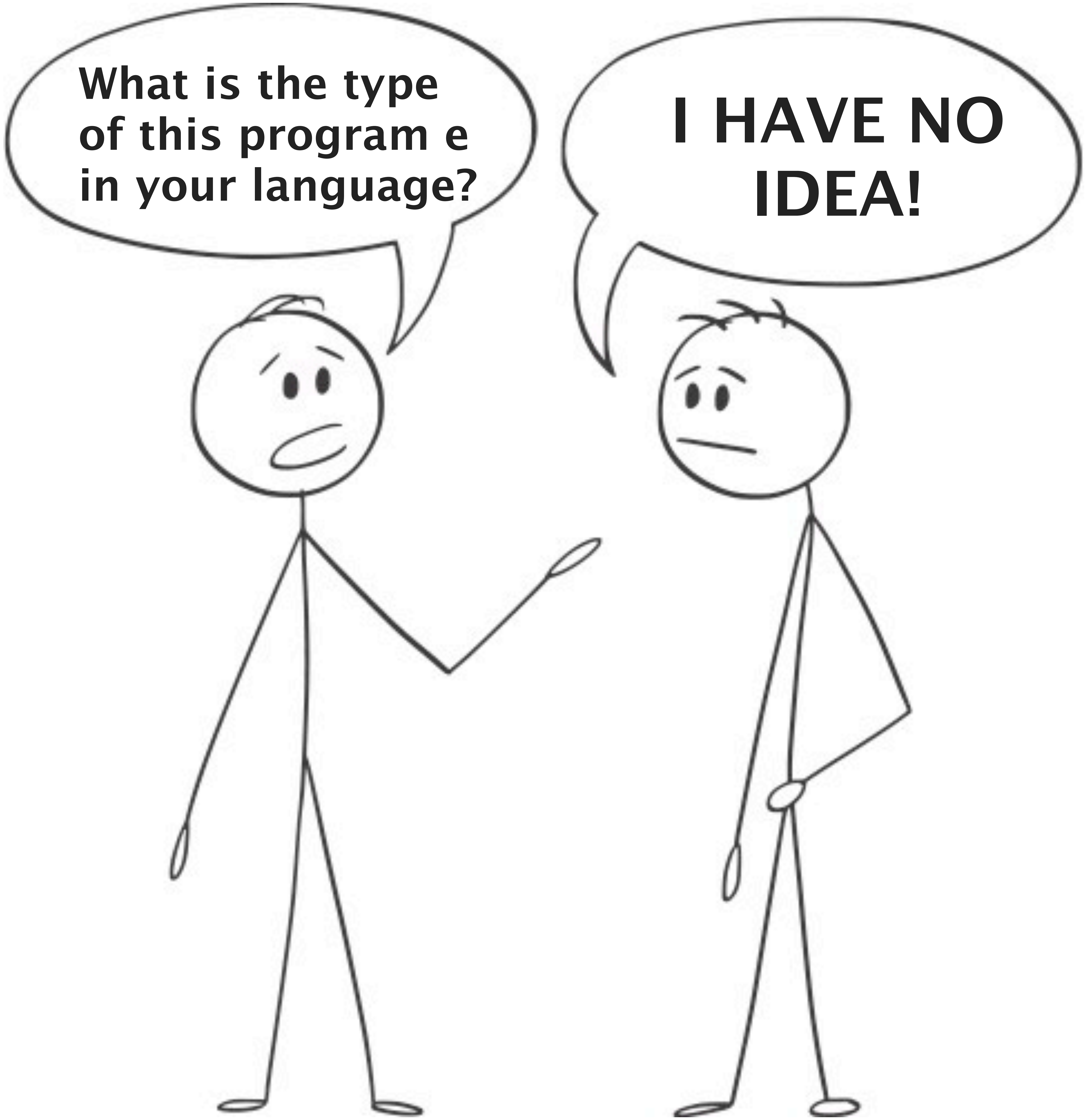


Need for Formal Type System

- **What is your language does not have a formal type system?**
- **Same problem at compile time**
- ***Bottom Line: You need a well-defined formal reference when implementing a compiler***

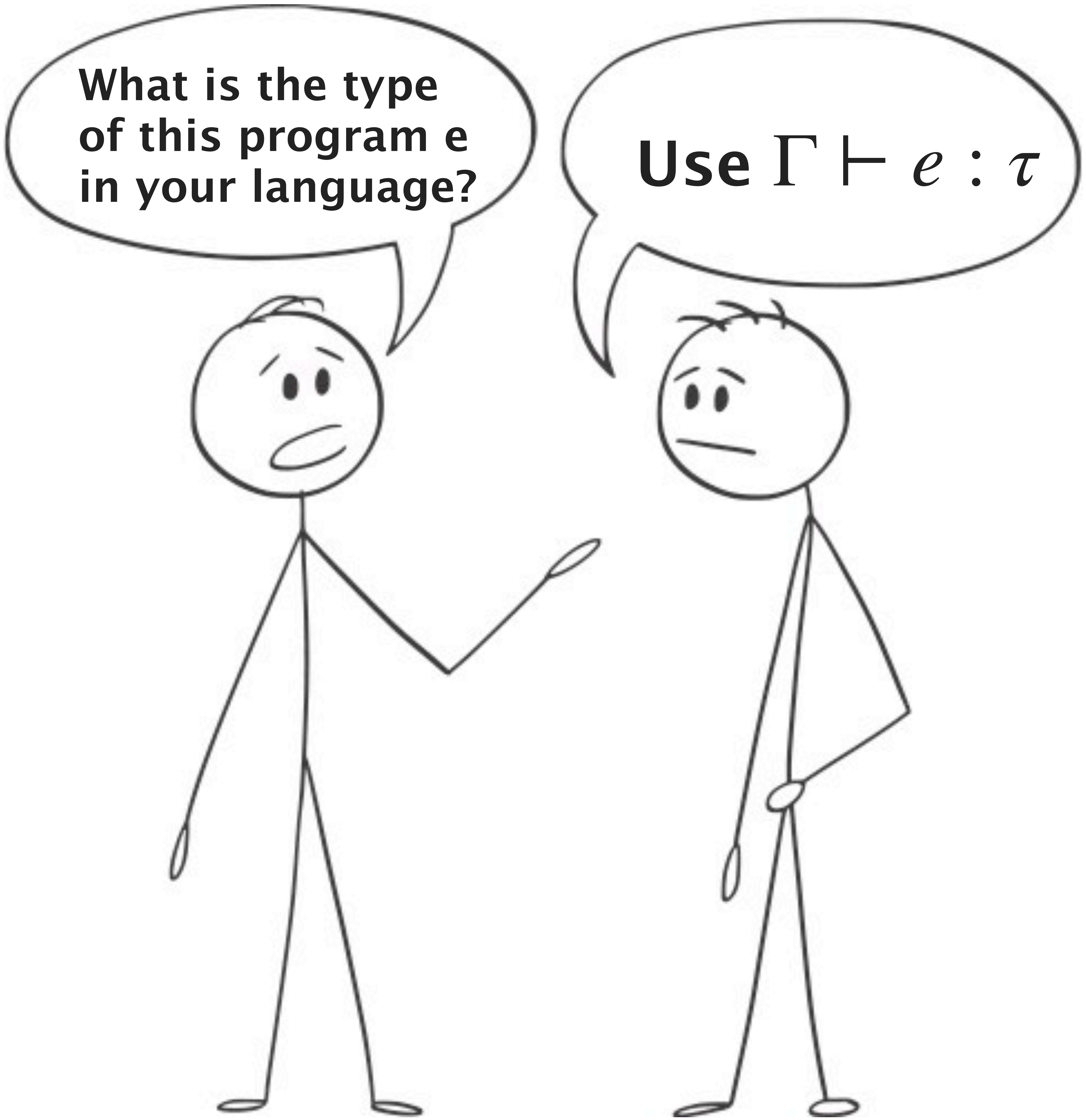
Need for Formal Type System

- What is your language does not have a formal type system?
- Same problem at compile time
- *Bottom Line: You need a well-defined formal reference when implementing a compiler*

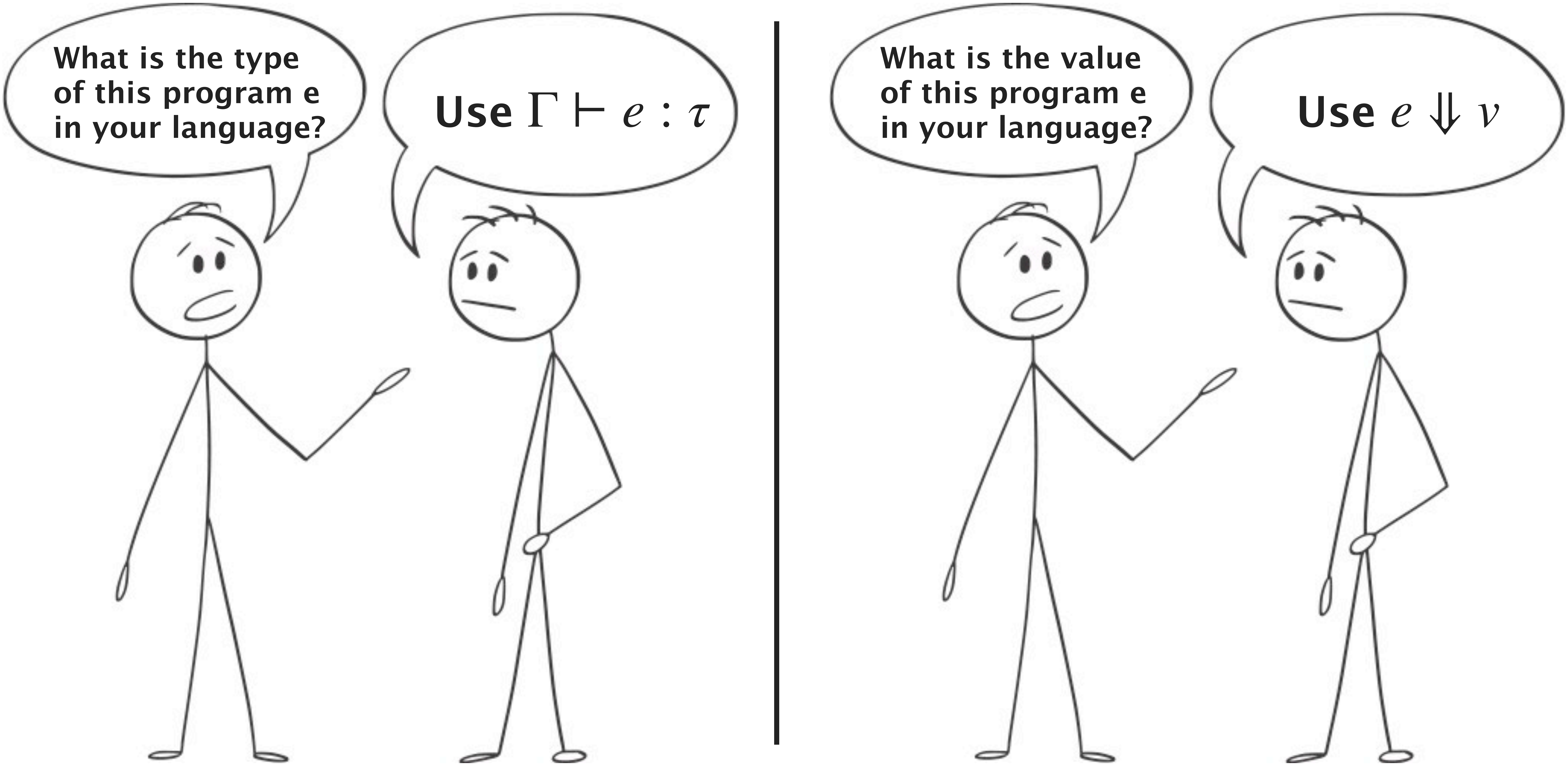


What Happens with a Good PL?

What Happens with a Good PL?



What Happens with a Good PL?



What Happens with Famous PLs?

- Designing a compiler requires building a type checker and interpreter
- How do you design a type checker without a formal specification of type system?
- *You make your own decisions!*
- Different compilers can make different decisions
- *Result: Same program executed with different compilers result in different values! REALLY BAD SITUATION*
- In a good PL, all compilers are designed using the formal spec

Lesson: How to Design a Good PL

- Write the formal syntax, type system, and semantics
- Implement the parser following the formal syntax
- Implement the type checker following the formal type system
- Implement the interpreter following the formal semantics
- *Can you still make mistakes? Yes, but the chances are much lower*
  (than if you were making decisions on the fly)

Bad PL Features

Bad PL Features

- Null Pointers
- Operator Overloading
- Type Casting
- I will explain what the issues are with these PL features

Why are Null Pointers Bad?

- Suppose a function hasan argument of type `int*`

- Can it be NULL? Depends on who's calling

- In a big codebase, it's very easy to lose track of which pointers can be NULL and which cannot

- This leads to segmentation faults

```
int foo(int* x) {
  // Can x be NULL?
  if (x == NULL) {
    ...
  }
  ...
}
```

Option Types are Better

- Don't ask the programmer to track which pointers can be NULL
- Instead let the type system (compiler) track this
- Much easier for the compiler to track
- Safer and automated as well
- No possibility of segmentation faults

```
int foo(Option<int*> x) {
    match x with
    | None -> ...
    | Some(x) -> ...
}
```

Type checker forces* programmer to do a NULL check

*Programmers are lazy; won't do anything unless you force them to!!

No Option ➜ Cannot be NULL

- If the argument has type int*, it cannot be called by NULL
- This can be easily enforced by the type system (OCaml does this!)
- Safe and automatic
- No possibility of segmentation faults

```
int foo(int* x) {
  // no need for NULL check
  ...
}

foo(NULL)  ❌
```

Foo cannot be called with NULL; type is int* not Option<int*>

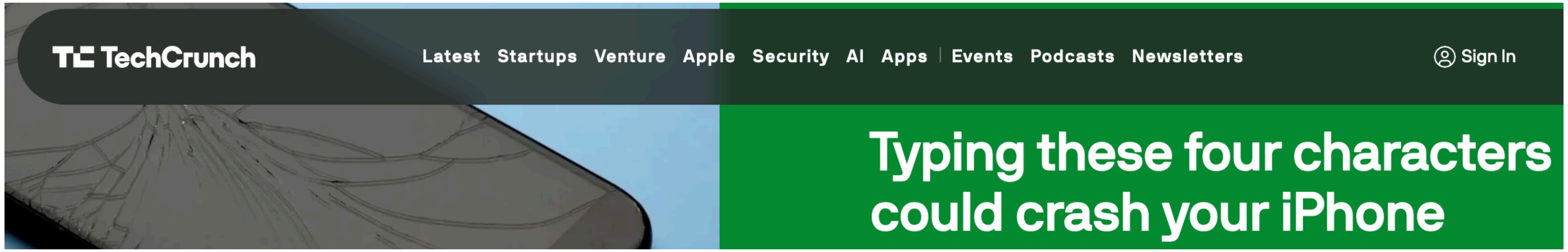
Famous NULL dereference bugs!

Debian User Forums
The Debian Project User and Community Forums

BUG: kernel NULL pointer dereference, address: 0000000000000038

CroCoder Home

The CrowdStrike and Microsoft Incident



Chromium Search Chromium Issues

Chromium > Blink > JavaScript > WebAssembly 40050993

A null pointer dereference has been discovered in V8 compiler which affects the latest version.

Why is Operator Overloading Bad?

- **Suppose we allowed adding integers and floats**

- **What's the issue? Let's see the typing rules**

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 + e_2 : \mathsf{float}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{float} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{float}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{float} \qquad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 + e_2 : \mathsf{float}}$$

- ***Problem: No longer a syntax directed type system!*** **So what?**

How would Type Checking Work?


```
let type_check ctx e tp =
  match e with
  | Add(e1, e2) ->
      if       (type_check ctx e1 Int && type_check ctx e2 Int && tp = Int) then true
      else if  (type_check ctx e1 Float && type_check ctx e2 Int && tp = Float) then true
      else if  (type_check ctx e1 Int && type_check ctx e2 Float && tp = Float) then true
      else if  (type_check ctx e1 Float && type_check ctx e2 Float && tp = Float) then true
      else false
```

- So many cases! But who cares?
- What’s the complexity of type checking?

How would Type Checking Work?


```
let type_check ctx e tp =
  match e with
  | Add(e1, e2) ->
      if        (type_check ctx e1 Int && type_check ctx e2 Int && tp = Int) then true
      else if   (type_check ctx e1 Float && type_check ctx e2 Int && tp = Float) then true
      else if   (type_check ctx e1 Int && type_check ctx e2 Float && tp = Float) then true
      else if   (type_check ctx e1 Float && type_check ctx e2 Float && tp = Float) then true
      else false
```

- So many cases! But who cares?
- What's the complexity of type checking? EXPONENTIAL!!!

Why is Type Casting Bad?

- What is type casting? An expression at one type can be used at a different type
- How does this look formally?

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2}$$

- What's the problem? To understand, we first need to understand the mathematical theorems
- We'll come back to this

Preservation Theorem

- The most important theorem for a good PL

- Theorem:
  Given a closed valid expression e such that $\cdot \vdash e : \tau$,
  if $e \Downarrow v$
  then $\cdot \vdash v : \tau$

- *Meaning:* A well-typed expression of a given type must evaluate to a value of the exact same type

- Let's see how we can prove this theorem! But before, let's revisit why type casting is bad

Revisiting Type Casting

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2}$$

- *Problem: Violates the preservation theorem!*
- **If $e \Downarrow v$, then irrespective of the type of v, the theorem is violated**
  - **If $v : \tau_1$ since $e : \tau_2$, the theorem fails on e**
  - **If $v : \tau_2$ since $e : \tau_1$, the theorem fails on e again**

Coming Back to Preservation

- **Theorem:**
  Given a closed valid expression e such that $\cdot \vdash e : \tau$,
  if $e \Downarrow v$
  then $\cdot \vdash v : \tau$

- **Suppose my language is defined as**

$$
\begin{aligned}
\langle expr \rangle ::= \quad & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ \langle expr \rangle\ \mathsf{then}\ \langle expr \rangle\ \mathsf{else}\ \langle expr \rangle \\
& \mid \langle expr \rangle\ \mathsf{op}\ \langle expr \rangle \mid n
\end{aligned}
$$

- **How do we prove preservation? Any thoughts?**

Proving for True / False

- **Theorem:**
  Given a closed valid expression true *such that* $\cdot \vdash \text{true} : \tau$,
  if $\text{true} \Downarrow v$
  then $\cdot \vdash v : \tau$

$$\frac{}{\cdot \vdash \text{true} : \text{bool}} \qquad \frac{}{\text{true} \Downarrow \text{true}}$$

$$\frac{}{\cdot \vdash \text{false} : \text{bool}} \qquad \frac{}{\text{false} \Downarrow \text{false}}$$

Proving for If Expressions

- **Theorem:**
  Given a closed valid expression such that $\cdot \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau$,
  if $\text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow v$
  then $\cdot \vdash v : \tau$

$$\frac{\cdot \vdash e : \text{bool} \qquad \cdot \vdash e_1 : \tau \qquad \cdot \vdash e_2 : \tau}{\cdot \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$$

$$\frac{e \Downarrow \text{true} \qquad e_1 \Downarrow v_1}{(\text{if } e \text{ then } e_1 \text{ else } e_2) \Downarrow v_1}$$

Using Mathematical Induction

$$\frac{\cdot \vdash e : \mathsf{bool} \qquad \cdot \vdash e_1 : \tau \qquad \cdot \vdash e_2 : \tau}{\cdot \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad \frac{e \Downarrow \mathsf{true} \qquad e_1 \Downarrow v_1}{(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \Downarrow v_1}$$

- **Observe e_1. We use the theorem on e_1. This is our inductive hypothesis: assume theorem holds on a smaller expression, prove for a bigger expression**
- **Theorem:**
  Given a closed valid expression e_1 such that $\cdot \vdash e_1 : \tau$,
  if $e_1 \Downarrow v_1$
  then $\cdot \vdash v_1 : \tau$

Using Mathematical Induction

$$\frac{\cdot \vdash e : \mathsf{bool} \qquad \cdot \vdash e_1 : \tau \qquad \cdot \vdash e_2 : \tau}{\cdot \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad \frac{e \Downarrow \mathsf{false} \qquad e_2 \Downarrow v_2}{(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \Downarrow v_2}$$

- **Now observe e_2. We use the same theorem on e_2. Applying the inductive hypothesis on e_2.**
- **Theorem:**
  Given a closed valid expression e_2 such that $\cdot \vdash e_2 : \tau$,
  if $e_2 \Downarrow v_2$
  then $\cdot \vdash v_2 : \tau$

More About Preservation Theorem

- The same principle applies for arithmetic operation expressions or floating-point expressions
- Things get more interesting when variables are involved
- *Homework: Prove this theorem for 'let' expressions:* let $x = e_1$ in e_2
- *Mathematical Induction: your BFF for proving theorems about programming languages!*

A Little Bit of Marketing

- Interested in learning more about these topics?
- I teach a graduate course CS 599 in spring which is all about good/bad programming languages for concurrency
(this course was all about sequential programming)
- Interested in research? I advise graduate/undergraduate students
- Just want to chat about PL? Please stop by my office anytime!

Conclusion

- Last lecture before midterm done!
- Do as much practice as possible!
- Midterm next week: October 17
- Thank you all!
- Any questions for me?