Type Systems

Introduction

1. Popular type of static analysis
   - Specified as part of the language
     - Built into compilers/interpreters for the language
   - Purpose: Reduce possibility of software bugs by checking for logic errors
     - Applying operation to operands that doesn't make sense
   - Checks for errors using a set of rules
     - Variables, expressions, functions

Type Systems

1. Most widely used for of static analysis
   - Part of nearly all mainstream languages
     - Important for quality

Type Systems Motivation

```
class T {
    int f(float a, int b, int[] c) {
        if (a) // expect boolean, got float
            return b;
        else
            return c; // expect int, got int[]
        }
}
```

Type Systems 2

1. Most widely used for of static analysis
   - Part of nearly all mainstream languages
     - Important for quality
   - Provides notation useful for describing static analyses:
     - Type checking, dataflow analysis, symbolic execution

What Is a Type?

1. A type is a set of values
   - Example in Java:
     - int is the set of all integers between -2^31 and (2^31)-1
     - double is the set of all double-precision floating point numbers
     - boolean is the set {true, false}

More Examples

1. Foo is the set of all objects of class Foo
   - List<Integer> is the set of all Lists of Integer objects
     - List is a type constructor
     - List acts as a function from types to types
   - int -> int is the set of functions taking an int as input and returning another int
     - e.g., increment, a function that squares a number, etc.

Abstraction

1. All static analyses use abstraction
   - Represent sets of concrete values as abstract values
2. Why?
   - Can't directly reason about infinite sets of concrete values (wouldn't guarantee termination)
   - Improces performance even in case of (large) finite sets
3. In type systems, the abstractions are called types
   - Reason about values based on properties they have in common

What Is a Type? 2

1. A type is an example of an abstract value
   - Represents a set of concrete values
2. In type systems:
   - Every concrete value is an element of some abstract value
     - Every concrete value has a type

A Simple Typed Language

1. Type language based on lambda calculus
   - (expression) e := v | x | e1 + e2 | e1 e2
   - (value) v := i | lambda x:t => e
   - (integer) i
   - (variable) x
   - (type) t := int | t1 -> t2
2. Example program

```
(
    lambda x:int => (x + 1)
) (42)
```

Programs and Types

Program	Type
(x:int => (x + x))	int -> int
(x:int => (x + x))(10)	int
42(x:int => (x + 5))	NONE
(x:int => (y:int => (x + y)))	int -> (int -> int)
(x:int => x) + 10	NONE

Next Steps

1. How do we analyze programs using type systems?
   - Type systems have a well-developed notation for performing these analyses
2. Notation for type systems
3. Properties of type systems
4. Describing other analyses using types notation

Notation for Inference Rules

1. Inference rules have the following form:
   - If (hypothesis) is true, then (conclusion) is true
2. Type checking computes via reasoning:

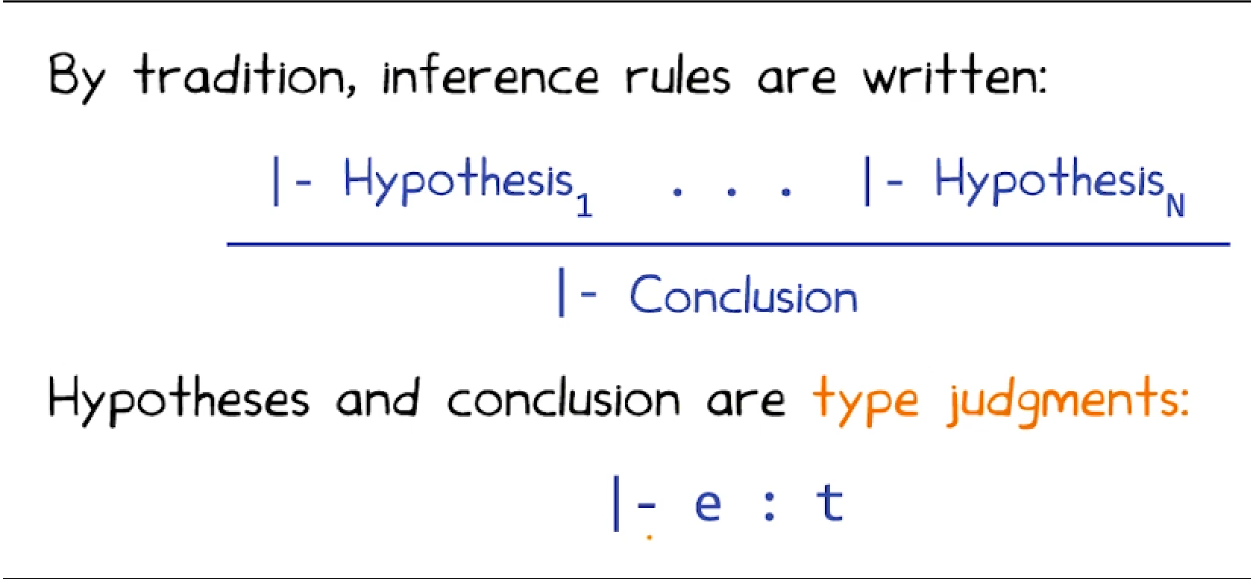
- If e1 is an int and e2 is a double, then e1*e2 is a double

3. We will develop a standard notation for rules of inference

From English to Inference Rule

1. Start with a simplified system and gradually add features
2. Building blocks:
   - Symbol ˆ is “and”
   - Symbol => is “if-then”
   - x : t is “x has type t”
3. If e1 is an int and e2 is a int, then e1+e2 is an int
   - (e1:int ˆ e2:int) => e1 + e2 : int
4. General form:
   - Hypothesis1 ˆ . . . ˆ hypothesisN => Conclusion

Notation for Inference Rules 2



By tradition, inference rules are written:

$$\frac{\vdash \text{Hypothesis}_1 \quad . \quad . \quad . \quad \vdash \text{Hypothesis}_N}{\vdash \text{Conclusion}}$$

Hypotheses and conclusion are type judgments:

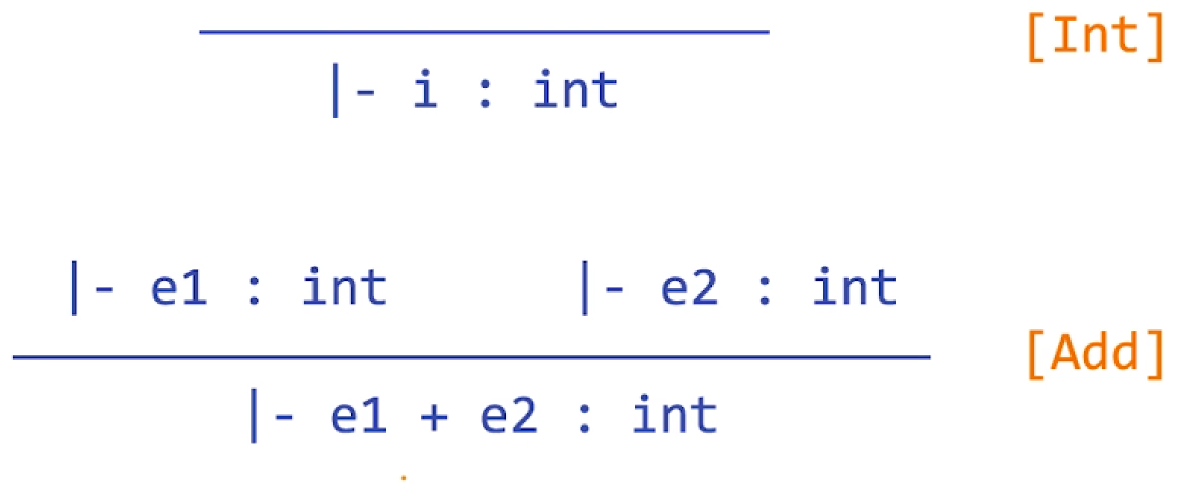
$$\vdash e : t$$

Notation for Inference Rules

1. |- means “it is provable that”

Rules for Integers

1. [Int] template rule is referred to as a schema
   - Because there are no hypotheses, it is an axion schema
2. [Add] rule is an inference rule schema because there are more than 0 hypotheses
3. Rules for Integers
   - Templates for how to type integers and sums
   - Filling in templates produces complete typings
   - Note that:
     – Hypotheses state facts about sub-expressions
     – Conclusions state facts about entire expression

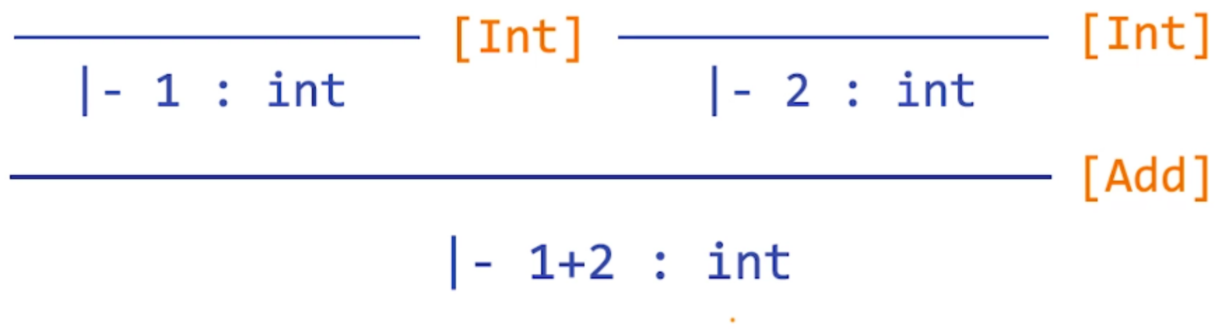


$$\dfrac{}{\texttt{|- i : int}}\ \texttt{[Int]}$$

$$\dfrac{\texttt{|- e1 : int} \qquad \texttt{|- e2 : int}}{\texttt{|- e1 + e2 : int}}\ \texttt{[Add]}$$

Notation for Inference Rules

Example: 1 + 2



$$\dfrac{\dfrac{}{\texttt{|- 1 : int}}\ \texttt{[Int]} \qquad \dfrac{}{\texttt{|- 2 : int}}\ \texttt{[Int]}}{\texttt{|- 1+2 : int}}\ \texttt{[Add]}$$

Example: 1 + 2

A Problem

1. We also want to apply typing rules for variables
   - Cannot axiomatically define the type of a variable
     - Don't want all variables to have to be of a specific type
2. Problem: What is the type of a variable reference?

A Solution

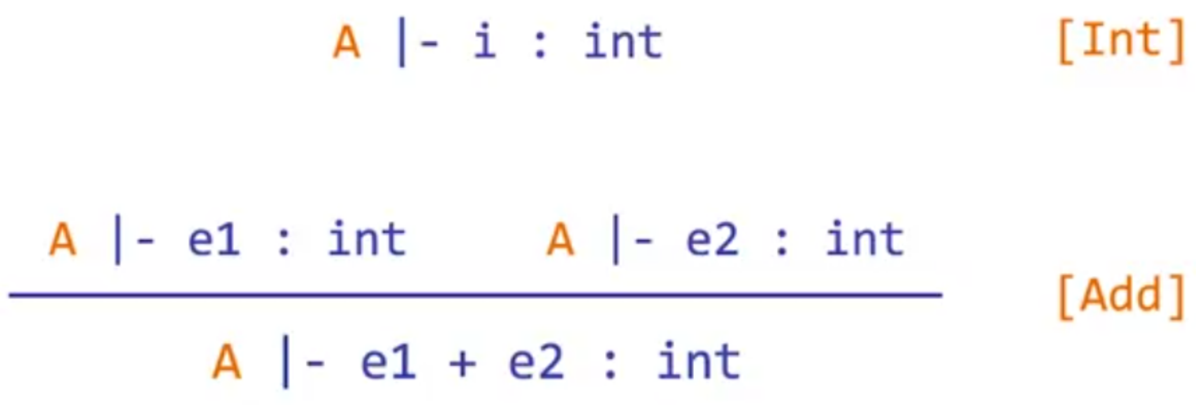
1. Put more information in the rules!
   - |- x : ?
2. An environment gives types for free variables
   - A variable is free in an expression if not defined within the expression; otherwise it is bound
   - An environment is a function from variables to types
   - May map variables to other abstract values in different static analyses

Type Environments

1. Let A be a function from variables to types
2. The sentence A |- e : t means:
   - "Under the assumption that variables have types given by A, it is provable that the expression e has type t."

Modified Rules

1. Modify the [Int] and [Add] rules to include the type environment before the pipe-hyphen symbol



$$A \mid\!- \; i : \mathtt{int} \qquad [\mathrm{Int}]$$

$$\frac{A \mid\!- \; e1 : \mathtt{int} \qquad A \mid\!- \; e2 : \mathtt{int}}{A \mid\!- \; e1 + e2 : \mathtt{int}} \qquad [\mathrm{Add}]$$

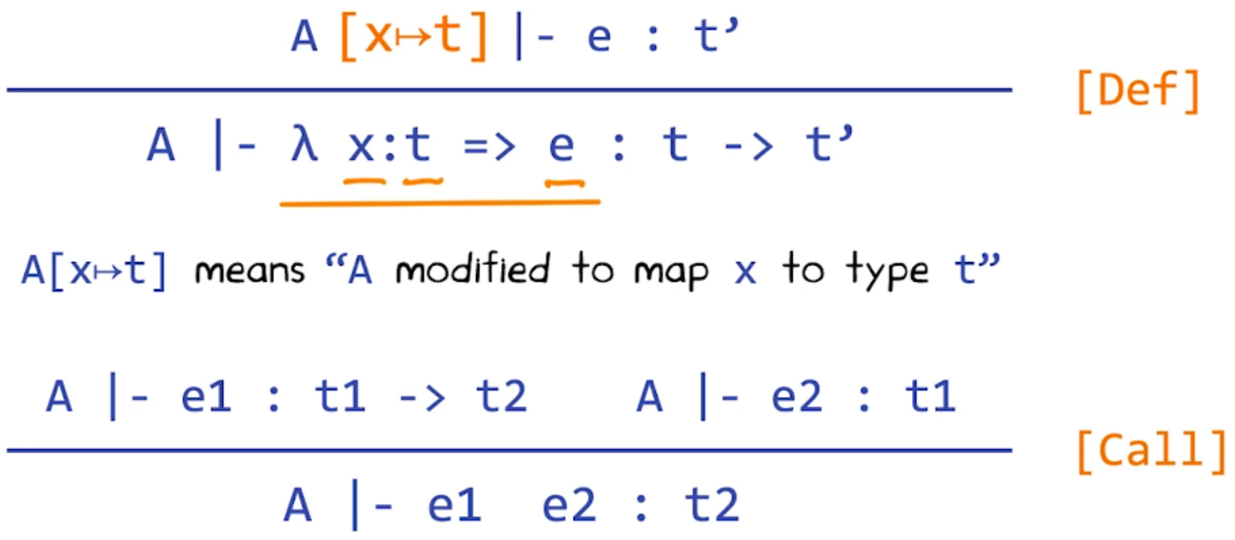
Modified Rules

A New Rule

1. And we can write new rules:
   -

Rules for Functions

1. Define rules for [Def] and [Call]
   - A[x->t] means "A modified to map x to type t"



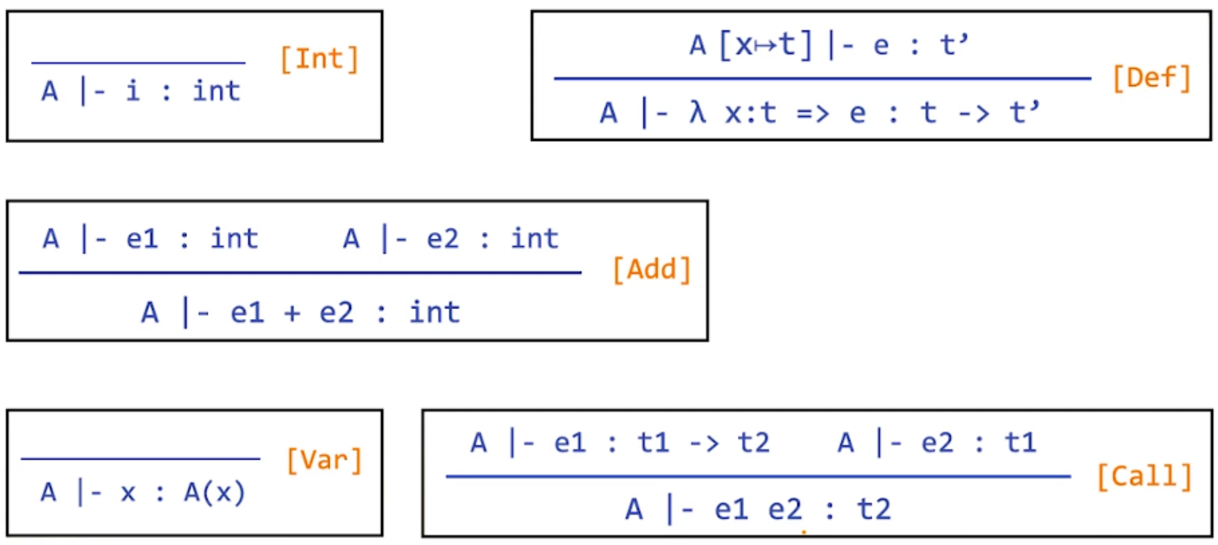
$$\frac{A\,[x \mapsto t]\ \vert\text{-}\ e : t'}{A\ \vert\text{-}\ \underline{\lambda\ \underline{x}:\underline{t} \Rightarrow \underline{e}}\ : t \to t'}\ [\text{Def}]$$

A[x↦t] means "A modified to map x to type t"

$$\frac{A\ \vert\text{-}\ e1 : t1 \to t2 \qquad A\ \vert\text{-}\ e2 : t1}{A\ \vert\text{-}\ e1\ \ e2 : t2}\ [\text{Call}]$$

Rules for Functions

All Rules Together



$$\frac{}{A\ \vert\text{-}\ i : int}\ [\text{Int}] \qquad \frac{A\,[x \mapsto t]\ \vert\text{-}\ e : t'}{A\ \vert\text{-}\ \lambda\ x:t \Rightarrow e : t \to t'}\ [\text{Def}]$$

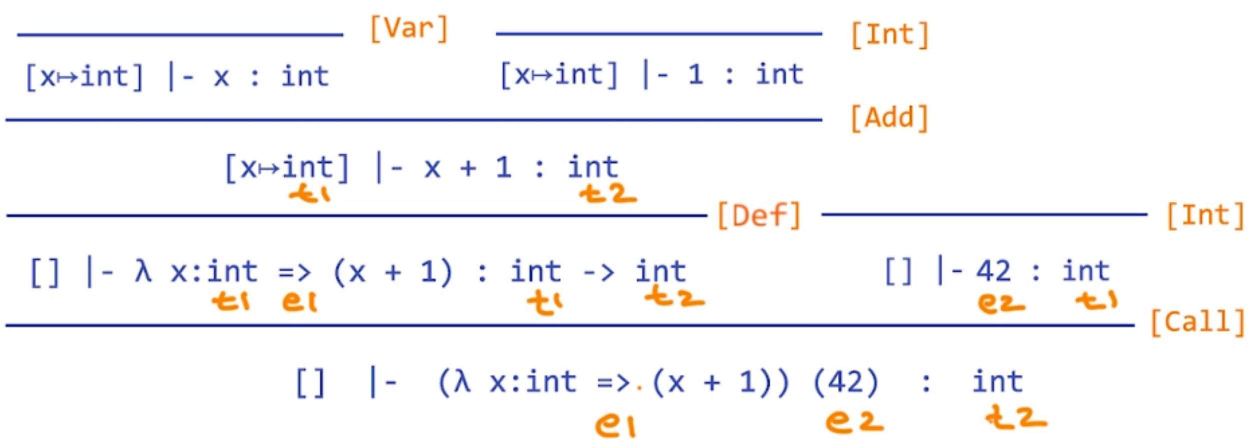
$$\frac{A\ \vert\text{-}\ e1 : int \qquad A\ \vert\text{-}\ e2 : int}{A\ \vert\text{-}\ e1 + e2 : int}\ [\text{Add}]$$

$$\frac{}{A\ \vert\text{-}\ x : A(x)}\ [\text{Var}] \qquad \frac{A\ \vert\text{-}\ e1 : t1 \to t2 \qquad A\ \vert\text{-}\ e2 : t1}{A\ \vert\text{-}\ e1\ e2 : t2}\ [\text{Call}]$$

All Rules

Type Derivations Example

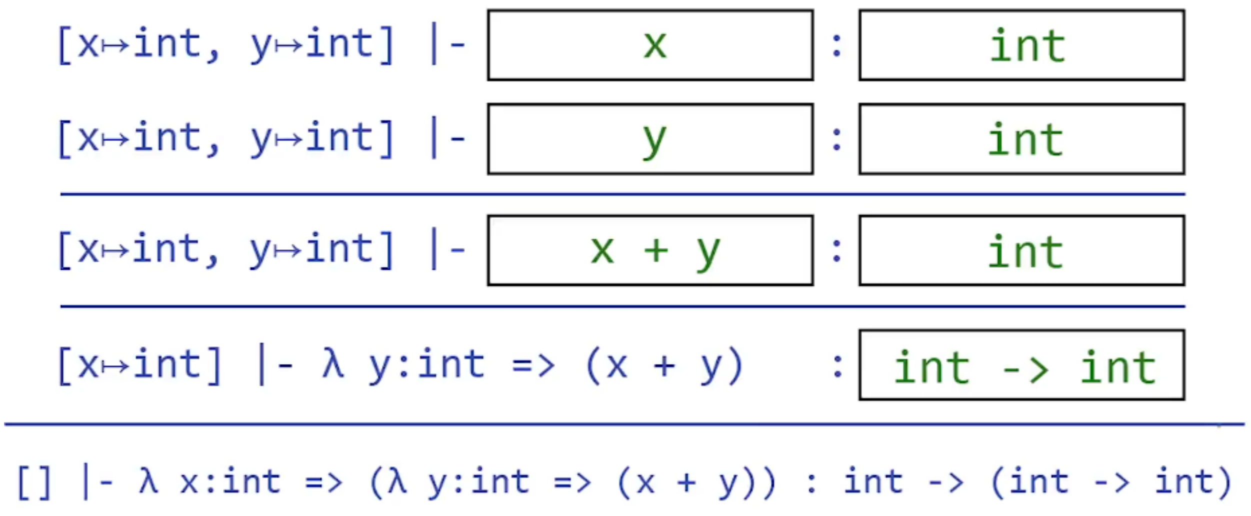
1. [] |- (x:int => (x + 1)) (42) : int



```
-------------------------- [Var]   -------------------------- [Int]
[x↦int] |- x : int                 [x↦int] |- 1 : int
---------------------------------------------------------------- [Add]
              [x↦int] |- x + 1 : int
                     t1          t2
------------------------------------------------ [Def]  ------------------ [Int]
[] |- λ x:int => (x + 1) : int -> int                   [] |- 42 : int
        t1      e1         t1     t2                          e2    t1
------------------------------------------------------------------------- [Call]
          [] |- (λ x:int => (x + 1)) (42) : int
                       e1              e2    t2
```

Derivation

Type Derivations



```
[x↦int, y↦int] |- x     : int
[x↦int, y↦int] |- y     : int
---------------------------------------
[x↦int, y↦int] |- x + y : int
---------------------------------------
[x↦int] |- λ y:int => (x + y)   : int -> int
------------------------------------------------
[] |- λ x:int => (λ y:int => (x + y)) : int -> (int -> int)
```

Derivation Quiz

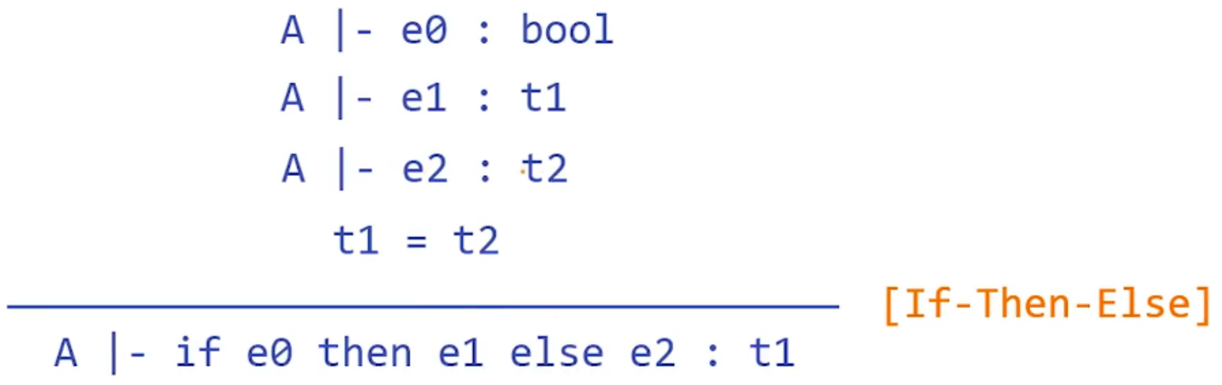
Back to the Original Example

1. Properties of type systems
    - What guarantees does this provide?
2. Algorithms for computing a type derivation if it exists

A More Complex Rule

1. Need to add a new rule for if-then-else statements
    - Adding a boolean type as well
    - t1 = t2 is called a side condition

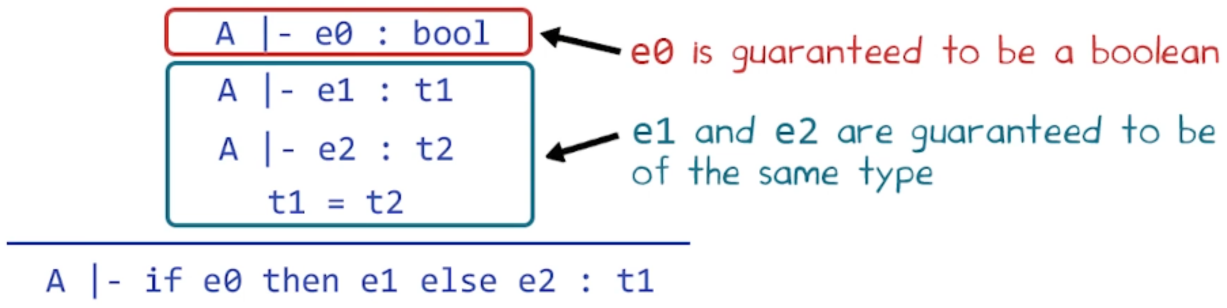
    - Additional constraint that must be satisfied in order for inference rule schema to be instantiated



If-Then-Else

Soundness

1. A type system is sound iff:
    - A |- e : t and
    - If A(x) = t', then x has a value v' in t', then e evalues to a value v in t


A |- e0 : bool
A |- e1 : t1
A |- e2 : t2
t1 = t2
A |- if e0 then e1 else e2 : t1
e0 is guaranteed to be a boolean
e1 and e2 are guaranteed to be of the same type

Soundness

Comments on Soundness

1. Soundness is extremely useful
    - Program type-checks -> no errors at runtimes
    - Verifices absence of a class of errors
2. This is a very strong guarantee
    - Verified property holds in all executions
    - "Well-typed programs cannot go wrong" -Robin Milner
3. Soundness comes at a price: false positives
    - Alternative: use unsound analysis
        - Reduces false positives
        - Introduces false negatives
    - Type systems are sound
        - But most bug finding analyses are not sound

Constraints

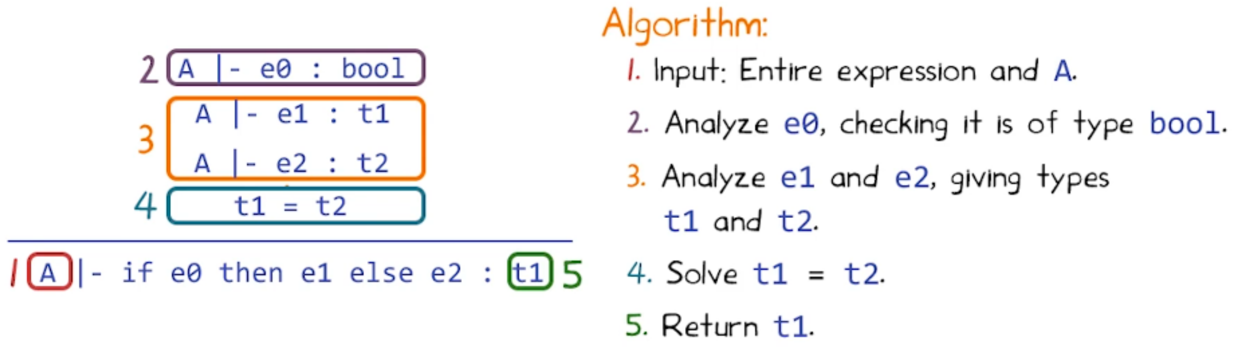
1. Many analyses have side conditions
   - Often constraints to be solved
   - All constraints must be satisfied
   - A separate algorithmic problem

Another Example

1. Consider a recursive function
   - f(x) = … f(e) …
2. If x: t1 and e: t2 then t2 = t1
   - Can be relaxed to t2 is a subset of t1
3. Recursive functions yield recursive constraints
   - Same with loops
   - How hard constraints are to solve depends on constraint language, details of application

Type-Checking Algorithm

1. Algorithm:
   - Input: Entire expression and A
   - Analyze e0, checking it is of type bool
   - Analyze e1 and e2, giving types t1 and t2
   - Solve t1 = t2
   - Return t1


2 A |- e0 : bool
3 A |- e1 : t1
A |- e2 : t2
4 t1 = t2
1 A |- if e0 then e1 else e2 : t1 5
Algorithm:
1. Input: Entire expression and A.
2. Analyze e0, checking it is of type bool.
3. Analyze e1 and e2, giving types t1 and t2.
4. Solve t1 = t2.
5. Return t1.

Global Type Checking Algorithm

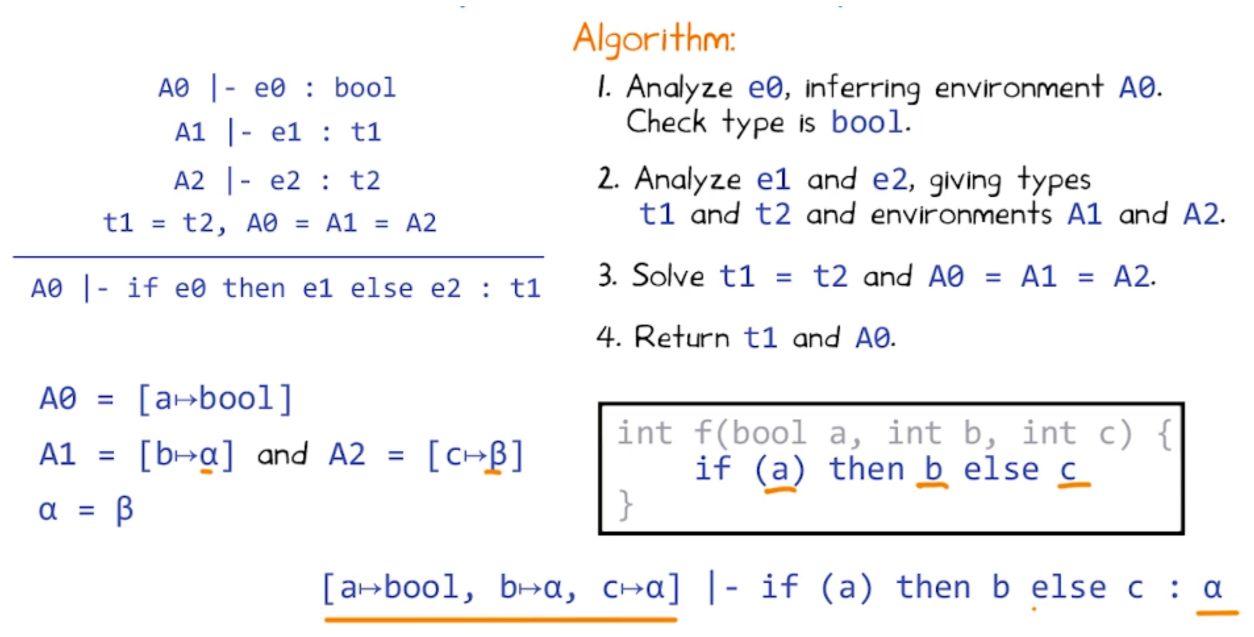
Global Analysis

1. Step 1 requires the overall environment A
   - Only then can we analyze subexpressions
   - This is global analysis
     - Requires the entire program
     - Or constructing a model of the environment

Local Analysis

1. Algorithm:
   - Analyze e0, inferring environment A0. Check type is bool
   - Analyze e1 and e2, giving types t1 and t2 and environments A1 and A2
   - Solve t1 = t2 and A0 = A1 = A2

    - Return r1 and A0
2. Also called compositional analysis or bottom-up analysis



$$\frac{\begin{array}{c}\texttt{A0 |- e0 : bool} \\ \texttt{A1 |- e1 : t1} \\ \texttt{A2 |- e2 : t2} \\ \texttt{t1 = t2, A0 = A1 = A2}\end{array}}{\texttt{A0 |- if e0 then e1 else e2 : t1}}$$

Algorithm:

1. Analyze e0, inferring environment A0. Check type is bool.
2. Analyze e1 and e2, giving types t1 and t2 and environments A1 and A2.
3. Solve t1 = t2 and A0 = A1 = A2.
4. Return t1 and A0.

A0 = [a↦bool]

A1 = [b↦α] and A2 = [c↦β]

α = β

```
int f(bool a, int b, int c) {
    if (a) then b else c
}
```

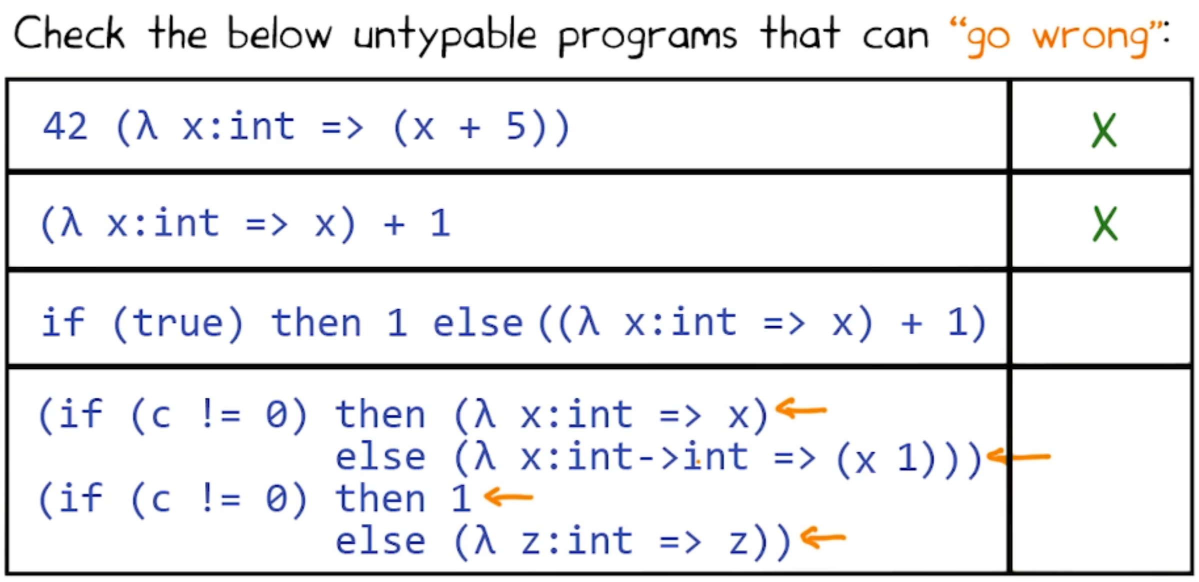
[a↦bool, b↦α, c↦α] |- if (a) then b else c : α

Local Type Checking Algorithm

Global vs Local Analysis

1. Global analysis:
    - Usually technically simpler than local analysis
    - May need extra work to model environments for unfinished programs
2. Local analysis:
    - More flexible in application
    - Technically harder: Need to allow unknown parameters, more side conditions

Properties of Type Systems



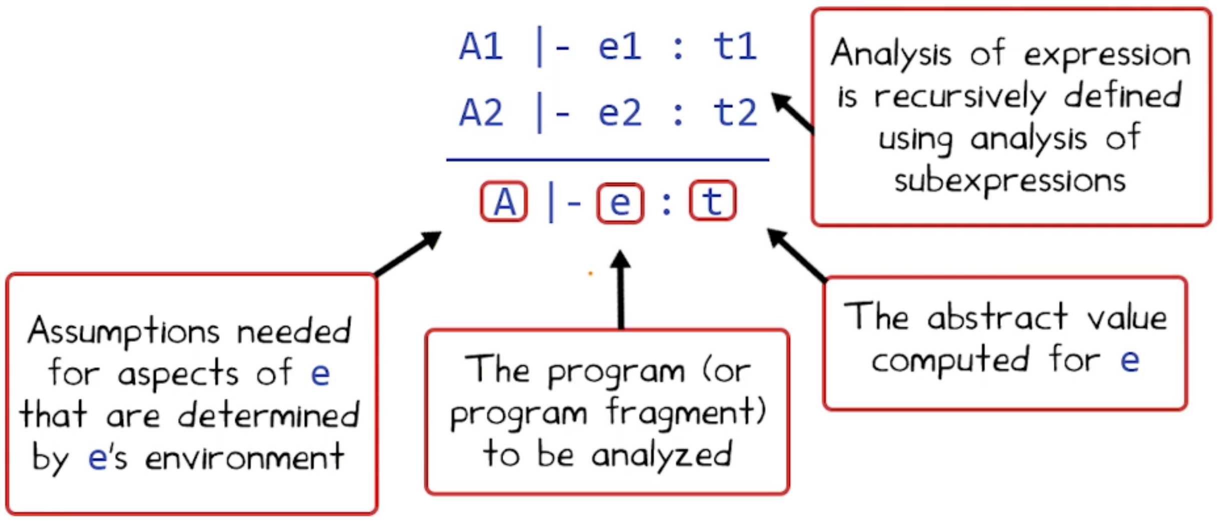
Check the below untypable programs that can "go wrong":

`42 (λ x:int => (x + 5))`	X
`(λ x:int => x) + 1`	X
`if (true) then 1 else ((λ x:int => x) + 1)`	

`(if (c != 0) then (λ x:int => x)`
`else (λ x:int->int => (x 1)))`
`(if (c != 0) then 1`
`else (λ z:int => z))`	

Type System Quiz

Static Analysis Using Type Rules



A1 |- e1 : t1
A2 |- e2 : t2
A |- e : t
Analysis of expression is recursively defined using analysis of subexpressions
Assumptions needed for aspects of e that are determined by e's environment
The program (or program fragment) to be analyzed
The abstract value computed for e

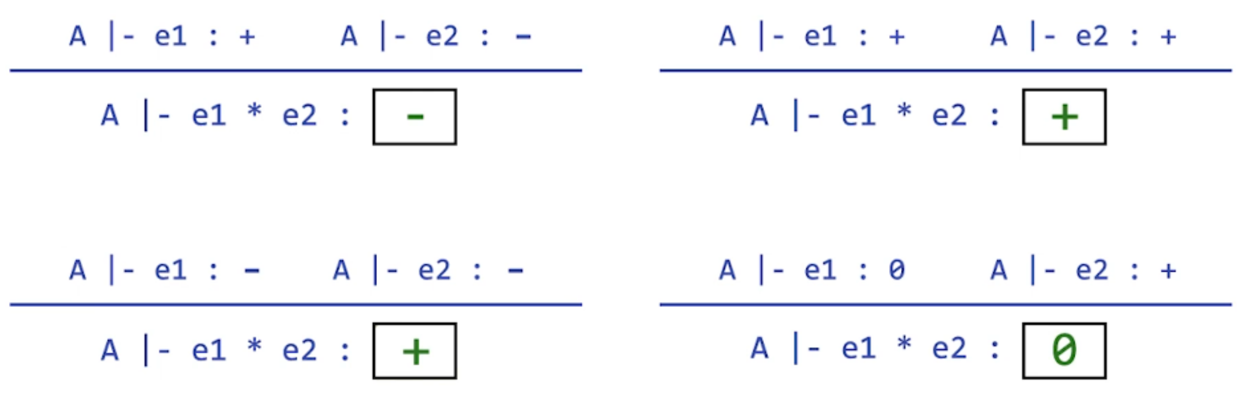
Static Analysis

An Example: The Rule of Signs

1. Goal: To estimate the sign of a numeric computation
2. Example: -3 * 4 = 12
3. Abstraction: - * + = -
4. Abstract Values

- − = {all positive integers}
- 0 = {0}
- − = {all negative integers}
- Environment A: Variables -> {+, 0, -}

Example Rules



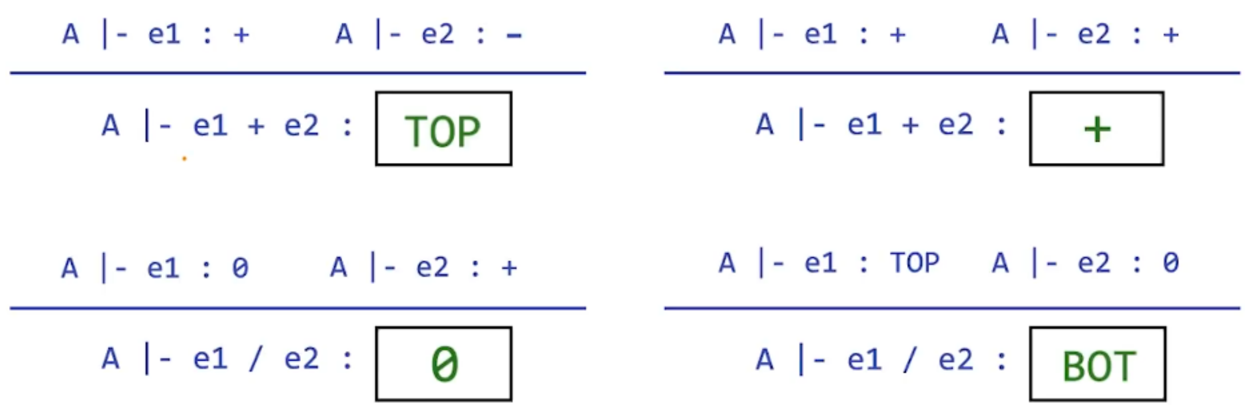
```
A |- e1 : +    A |- e2 : -          A |- e1 : +    A |- e2 : +
--------------------------          --------------------------
   A |- e1 * e2 : [ - ]                A |- e1 * e2 : [ + ]

A |- e1 : -    A |- e2 : -          A |- e1 : 0    A |- e2 : +
--------------------------          --------------------------
   A |- e1 * e2 : [ + ]                A |- e1 * e2 : [ 0 ]
```

Sign Quiz

Another Problem

1. If e1 is positive and e2 is negative, then e1 + e2 is. . .
   - We don't have an abstract value that covers this case
2. Solution:
   - Add abstract values to ensure closure under all operations
     - ∗ = {all positive integers}
     - 0 = {0}
     - ∗ = {all negative integers}
     - TOP = {all integers}
     - BOT = {}

More Example Rules



```
A |- e1 : +    A |- e2 : -          A |- e1 : +    A |- e2 : +
--------------------------          --------------------------
   A |- e1 + e2 : [ TOP ]              A |- e1 + e2 : [ + ]

A |- e1 : 0    A |- e2 : +          A |- e1 : TOP  A |- e2 : 0
--------------------------          --------------------------
   A |- e1 / e2 : [ 0 ]                A |- e1 / e2 : [ BOT ]
```

Sign Quiz 2

Flow Insensitivity

1. In the if-then-else example, the subexpressions are independent of each other
    - Flow-insensitive analysis: analysis is independent of the ordering of sub-expressions
        - Analysis results unaffected by permuting statements
    - Type systems are generally flow-insensitive

Comments on Flow Insensitivity

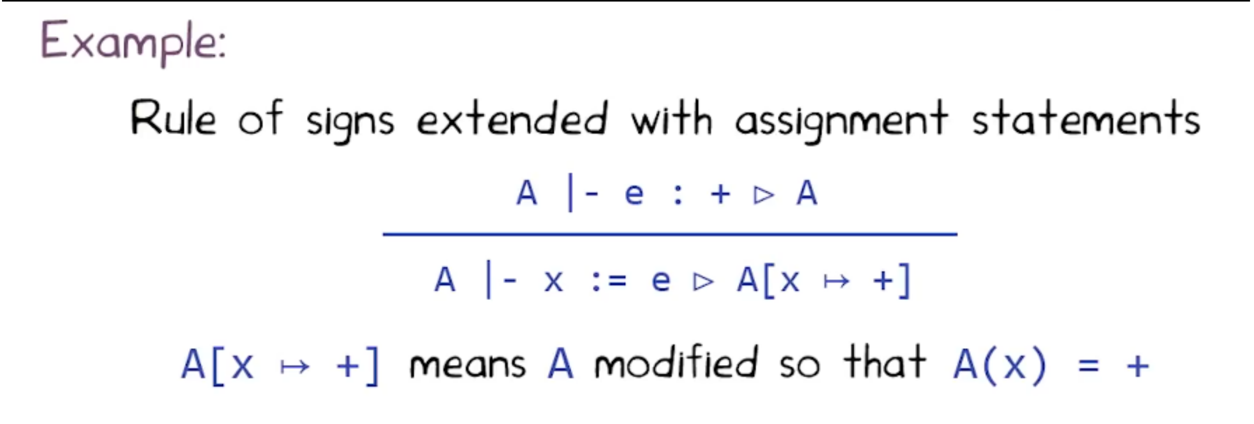
1. No need for modeling a separate state for each subexpression
2. Flow insensitive analyses are often very efficient and scalable
3. But can be imprecise…

Flow Sensitivity

1. Rules produce new environments, and analysis of a subexpression cannot happen until its environment is available
    - Flow-sensitive analysis: Analysis of subexpressions ordered by environments
        - Analysis result depends on order of statements
    - Dataflow analysis is example of flow-sensitive analysis

Comments on Flow Sensitivity

1. Example: Rule of signs extended with assignment statements
    - Flow-sensitive analysis can be expensive
        - Each statement has own model of state
        - Polynomial cost increase over flow-insensitive



Example:

Rule of signs extended with assignment statements

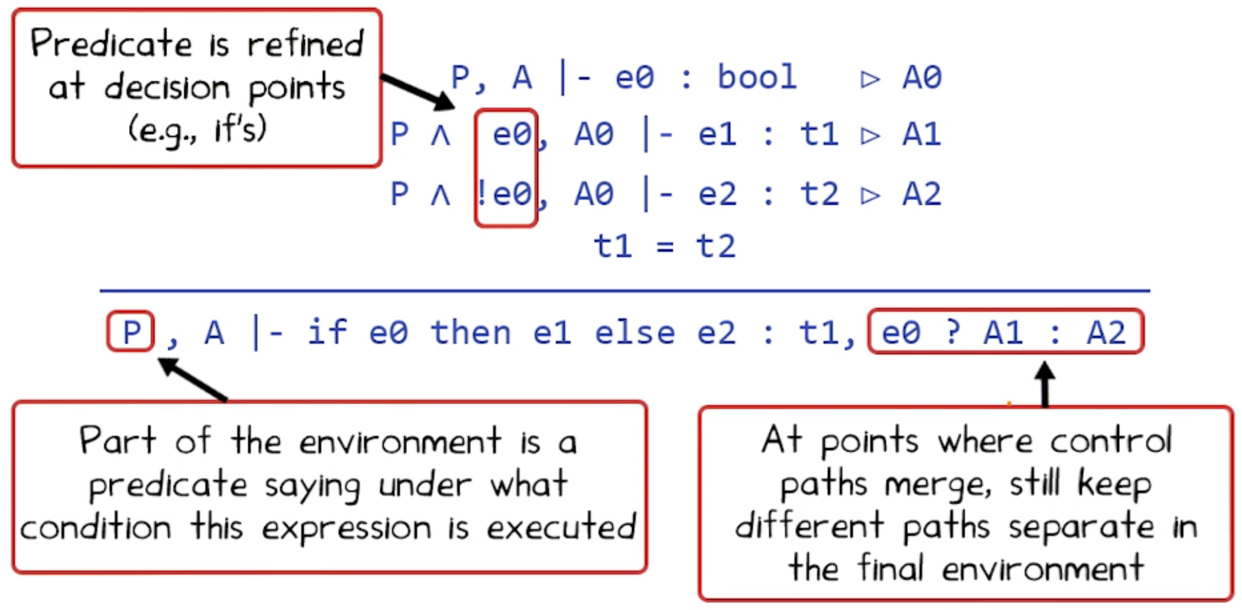
$$\frac{A \vdash e : + \triangleright A}{A \vdash x := e \triangleright A[x \mapsto +]}$$

$A[x \mapsto +]$ means A modified so that $A(x) = +$

Flow Sensitivity

Path Sensitivity

1. Path sensitivity eliminates the issue of a failed type check in dead code
    - Dead code: Code which is logically unreachable
2. Path sensitivity allows us to eliminate a class of false positives at the cost of increased complexity

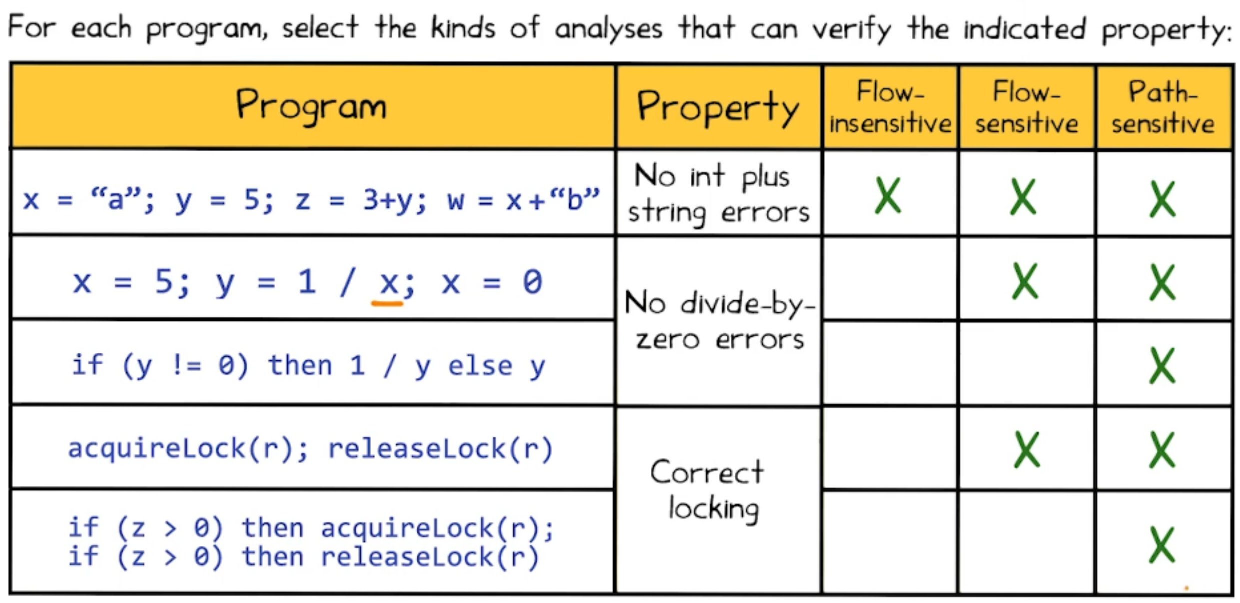

Predicate is refined at decision points (e.g., if's)
P, A |- e0 : bool ▷ A0
P ∧ e0, A0 |- e1 : t1 ▷ A1
P ∧ !e0, A0 |- e2 : t2 ▷ A2
t1 = t2
P, A |- if e0 then e1 else e2 : t1, e0 ? A1 : A2
Part of the environment is a predicate saying under what condition this expression is executed
At points where control paths merge, still keep different paths separate in the final environment

Path Sensitivity

Comments on Path Sensitivity

1. Symbolic execution is an example
    - Path-sensitive analyses are also flow-sensitive
2. Can be expensive by a necessary evil
    - Exponential number of paths to track
3. Often implemented with backtracking
    - Explore one path
    - Backtrack to a decision point, explore another path

Flow & Path Sensitivity



For each program, select the kinds of analyses that can verify the indicated property:

Program	Property	Flow-insensitive	Flow-sensitive	Path-sensitive
`x = "a"; y = 5; z = 3+y; w = x+"b"`	No int plus string errors	X	X	X
`x = 5; y = 1 / x; x = 0`	No divide-by-zero errors		X	X
`if (y != 0) then 1 / y else y`				X
`acquireLock(r); releaseLock(r)`	Correct locking		X	X
`if (z > 0) then acquireLock(r);` `if (z > 0) then releaseLock(r)`				X

Flow & Path Sensitivity Quiz

Summary

1. Very rough taxonomy:
    - Type systems = flow-insensitive
    - Dataflow analysis = flow-sensitive
    - Symbolic execution = path-sensitive
2. Lines have been blurred
    - Many flow-sensitive type systems and path-sensitive dataflow analyses in research literature

Conclusion

1. What is a type
2. Computing types of programs using type rules
3. Properties of type systems: soundness, incompleteness, global vs. local type checking
4. Describing other analyses using types notation
5. Classifying analyses: flow-insensitive vs. flow-sensitive vs. path-sensitive