

## Type Systems

Mayur Naik



```
s.close()
for i in range(1, 1000):
    attack()

import os
print os.getcwd()
print id = os.fork()
def attack():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #pid = os.fork()
    s.connect((sys.argv[1], sys.argv[2]))
    print ">> GET /" + sys.argv[1]
    s.send("GET /" + sys.argv[1])
    s.send("Host: " + sys.argv[2])
    s.close()
for i in range(1, 1000):
```

This module introduces a popular kind of static analysis called type systems.

Type systems are often specified as part of the programming language, and built into compilers or interpreters for the language.

The main purpose of a type system is to reduce the possibility of bugs by checking for logic errors. A common example of such an error is applying an operation to operands that does not make sense, such as adding an integer to a string in a Java program.

A type system checks for such errors using a collection of rules which assign types to different parts of the program, such as variables, expressions, and functions.

We will learn about the notation for type systems, various useful properties of type systems, and how type systems can even be used for describing other kinds of static analyses.

## LESSON

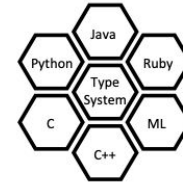
### Introduction

## LESSON

### Motivation and Background

## Type Systems

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



Type systems are the most widely used form of static analysis.

They are used in nearly all mainstream programming languages; one reason for this is that they are widely recognized as being important for maintaining the quality of programs.

CIS 547 - Software Analysis

## Motivation

```
1: class T {
2:   int f(float a, int b,
3:     int[] c) {
4:     if (a)
5:       return b;
6:     else
7:       return c;
8:   }
9: }
```

File T.java

```
prompt$ javac T.java
T.java:4: error: incompatible types
    if (a)
       ^
    required: boolean
    found:    float
T.java:7: error: incompatible types
    return c;
           ^
    required: int
    found:   int[]
2 errors
```

Penn Engineering Property of Penn Engineering | 5

Let's look at an example of the kind of errors that a type system typically catches.

Consider the following Java program which defines a function `f` in class `T` in a file named `T.java`. When we compile this program using the java compiler, it produces two type errors:

The first error states that the `if`-condition on line 4 expects its argument `a` to be of type `boolean`, but it is instead of type `float`.

The second error states that the return result `c` of the function `f` on line 7 is expected to be of type `int`, since the specified return type of function `f` is `int`, but it instead has type `integer array`.

Java's type system prevents logic errors like these which could cause the program to produce undesired results.

CIS 547 - Software Analysis

## Type Systems

- Most widely used form 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, ...

Penn Engineering Property of Penn Engineering | 6

Besides being a static analysis in their own right, type systems also provide notation which is useful for describing a variety of different static analyses, such as type checking, dataflow analysis, symbolic execution, and more.

CS 547 - Software Analysis

## What Is a Type?

- A **type** is a **set of values**
- Examples 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**}

Penn Engineering Property of Penn Engineering | 7

To discuss type systems, we must first ask ourselves, what do we mean by a “type”? It is easy to label the number 2 an integer and the number 1.125 a floating-point number, but it isn’t necessarily obvious how to define what a “type” is.

To put it simply, a “type” is a set of values that constitute that type. For example, in Java, the type “int” is the set of all integers (at least, those integers that can be represented in 32 bits of data); the “double” type is the set of real numbers that can be represented in a certain 64-bit IEEE standard, and the “boolean” type is the set consisting of the values “true” and “false.”

CS 547 - Software Analysis

## More Examples

- **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.

Penn Engineering Property of Penn Engineering | 8

Moving away from primitive types in Java, we can come up with more complex examples.

If we have a class named Foo, then the Foo type is the set of all objects of class Foo.

List<Integer> is the set of all List objects which contain Integer objects. Note that List is a type constructor: it can be treated as a function that takes one type (in this case, the Integer type) and maps it to another type (in this case, the type of Lists of Integers).

We can also have function types. For example, int arrow int is the set of all functions that take an int as their argument and return an int as their result. Some examples of functions of this type would be the increment operation, a function that squares its input, and many others.

CS 547 - Software Analysis

## SEGMENT

# Types and Abstraction

Penn Engineering

Property of Penn Engineering | 9

CS 547 - Software Analysis

## Abstraction

- All static analyses use **abstraction**
  - Represent sets of concrete values as abstract values
- **Why?**
  - Can't directly reason about infinite sets of concrete values (wouldn't guarantee **termination**)
  - Improves **performance** even in case of (large) finite sets
- In type systems, the abstractions are called **types**

Penn Engineering

Property of Penn Engineering | 10

Abstractions are an integral part of all forms of static analysis. Recall that in dataflow analysis, we abstract the control flow of programs, and in pointer analysis, we additionally abstract the heap by lumping together objects allocated at the same allocation statement in the program.

Why do we do this? The reason for abstracting program properties goes back to the undecidability of static analysis: we cannot reason directly about the infinite set of all possible concrete values (if we tried to, we have no guarantee that our analysis would terminate). Additionally, even for finite sets of concrete values, abstraction allows us to improve the performance of our analysis.

The abstractions used in type systems are the types themselves: each type represents a large, possibly infinite set of concrete values. By abstracting away these large sets, we enable ourselves to reason about the values based on properties they have in common.

CS 547 - Software Analysis

## What Is a Type?

- A type is an example of an **abstract value**
  - Represents a **set of concrete** values
- **In type systems:**
  - Every concrete value is an element of some abstract value
    - => every concrete value has a type

Penn Engineering Property of Penn Engineering | 11

A type is an example of what is known as an abstract value in that it represents a set of concrete values: the Java type `int` represents all integers representable with 32 bits, and the `List` type represents the set of all `List` objects.

In type systems, every concrete value must be an element of some abstract value; therefore, every concrete value has a type.

CS 547 - Software Analysis

## The Next Steps

- Notation for Type Systems
- Properties of Type Systems
- Describing Other Analyses Using Types Notation

Penn Engineering Property of Penn Engineering | 12

Now that we have a better understanding of what types are, we will learn how to compute with them: that is, how to actually analyze programs using type systems.

Type systems have a well-developed notation for expressing these sorts of analyses. In fact, a major part of defining a type system involves defining a set of rules showing how to check that a given program has a given type.

In the next lesson, we will provide the notation for type systems, which consists of a set of rules that can be used to check that a given program has a given type. Later on, we will discuss important properties of type systems, including how to infer the type automatically for a given program. Finally, we will see how to use the notation of types to describe other static analyses.

## LESSON

### A Simple Typed Language

## SEGMENT

### The Lambda Calculus

CS 547 - Software Analysis

## The Lambda Calculus

(expression)	$e$	$:=$	$v$	$ $	$x$	$ $	$e_1 + e_2$	$ $	$e_1 e_2$
(value)	$v$	$:=$	$i$	$ $	$\lambda x:t \Rightarrow e$				
(integer)	$i$								
(variable)	$x$								
(type)	$t$	$:=$	$int$	$ $	$t_1 \rightarrow t_2$				

Example Program:

```

(
  λ x:int => (x + 1)
) (42)

```

Penn Engineering Property of Penn Engineering | 15

Let's introduce a simple typed language based on the lambda calculus. We will use this language to provide concrete examples for the abstract principles of type systems.

Each expression in this language ultimately evaluates to one of two kinds of values  $v$ : integers (denoted here by  $i$ , but which may be any integer value) and functions denoted as follows [point to "lambda  $x:t \Rightarrow e$ "], lambda  $x$  of type  $t$  yields  $e$  (where " $x$  of type  $t$ " is represented by  $x$  followed by a colon followed by  $t$ , and "yields" is represented by an  $=$  sign followed by a  $>$  sign). This notation means a function which takes an argument  $x$  of type  $t$ , and returns the result of computing expression  $e$ .

Both kinds of values -- integers and functions -- can be stored in variables denoted by  $x$ . (Other identifiers may be used for variables as well.)

The plus operation requires that both  $e_1$  and  $e_2$  be integer-valued expressions. Our type system will ensure that this requirement is indeed met in all executions of a program in this language; in other words, that  $e_1$  and  $e_2$  can never evaluate to a function value.

Similarly, this operation [point to  $e_1 e_2$ ] denotes a call to a function  $e_1$  with argument  $e_2$ , and there is a similar requirement on the kinds of values that  $e_1$  and  $e_2$  might evaluate to. Our type system will also ensure this requirement.

To meet these requirements, the system consists of two types, corresponding to the

two kinds of values in our language: an  $int$  type, which is the type of all integer values, and a function type denoted by  $t_1 \rightarrow t_2$  [point to  $t_1 \rightarrow t_2$ ], where the arrow is represented by a hyphen followed by a greater-than sign) which is the type of functions.

Note that the  $int$  type is a base type, whereas function type is a compound type, as we will illustrate using an example program in this language.

This example defines a function that takes an integer  $x$  as an argument and returns  $x+1$ . The type of this expression is  $int \rightarrow int$  [handwrite this]. Then this function is applied to the integer 42, yielding the integer 43.



CS 547 - Software Analysis

## QUIZ: Programs and Types

Write the type of each program, or NONE if it is not typeable:

Program	Type
$\lambda x:\text{int} \Rightarrow (x + x)$	
$(\lambda x:\text{int} \Rightarrow (x + x)) (10)$	
$42 (\lambda x:\text{int} \Rightarrow (x + 5))$	
$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$	
$(\lambda x:\text{int} \Rightarrow x) + 10$	

Penn Engineering Property of Penn Engineering | 16

{QUIZ SLIDE}

To check your understanding of this language, let's do a quiz. Here we've listed five short programs:

- lambda x of type int yielding x plus x
- lambda x of type int yielding x plus x, applied to 10
- 42 applied to lambda x of type int yielding x plus 5
- lambda x of type int yielding (lambda y of type int which in turn yields x plus y)
- lambda x of type int yielding x, plus 10

Write the type of each program in the box beside it, or NONE if it is not typeable. Note that, in our type system, each typeable program is guaranteed to have a unique type, so there will not be any ambiguity in the type of these expressions; please use parentheses to disambiguate the type if needed.

CS 547 - Software Analysis

## QUIZ: Programs and Types

Write the type of each program, or NONE if it is not typeable:

Program	Type
$\lambda x:\text{int} \Rightarrow (x + x)$	int -> int
$(\lambda x:\text{int} \Rightarrow (x + x)) (10)$	int
$42 (\lambda x:\text{int} \Rightarrow (x + 5))$	NONE
$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$	int -> (int -> int)
$(\lambda x:\text{int} \Rightarrow x) + 10$	NONE

Penn Engineering Property of Penn Engineering | 17

{SOLUTION SLIDE}

The first program, lambda x of type int yields x plus x, is a function that takes an integer and yields the expression x plus x (it is a function that doubles its input). The type of this expression is therefore int arrow int.

The second program takes the function in the first program and applies it to the number 10. The result of this program is 20, which has the type int.

The third program is not typeable as it violates a requirement of function calls, namely, that the first expression in the pair must be a function, whereas here it is an integer 42.

The next program is a bit trickier: it takes an integer x and returns a function that takes a number and adds it to x. So the type for the whole expression is the compound type int arrow (int arrow int), where we put parentheses around the latter two ints to disambiguate the type. (Putting parentheses around the first two ints would be the type of a function which takes a function from int to int, and returns an int.)

The last program is not typeable as it violates a requirement of the plus operation, namely, that both its operands be integers, whereas here one of the operands is a function.

CIS 547 - Software Analysis

**SEGMENT**

Typing Rules: Part I

Penn Engineering

Property of Penn Engineering | 18

CIS 547 - Software Analysis

**Notation for Inference Rules**

- Inference rules have the following form:  
*If (hypothesis) is true, then (conclusion) is true*
- Type checking computes via reasoning:  
*If e1 is an int and e2 is a double, then e1\*e2 is a double*
- We will develop a standard notation for inference rules

Penn Engineering

Property of Penn Engineering | 19

If you've studied formal logic, you may be familiar with logical implications: "if-then" statements. Inference rules in type systems are "if-then" statements which are used to reason about the types of program fragments, such as variables, expressions, and functions.

A rule of inference takes the form, "If (hypothesis) is true, then (conclusion) is true," where the hypothesis and the conclusion are logical formulas.

An example of an inference rule that might be used in a type system is: "if e1 is an int and e2 is a double, then e1 \* e2 is a double."

Now let's start developing a standard notation for expressing inference rules.

CS 547 - Software Analysis

## From English to Inference Rule

- Start with a simplified system and gradually add features
- Building blocks:
  - Symbol  $\wedge$  is “and”
  - Symbol  $\Rightarrow$  is “if-then”
  - $x : t$  is “x has type t”

Penn Engineering Property of Penn Engineering | 20

To build this notation, we'll begin with a few small building blocks and then gradually add features as needed.

The building blocks we will use are:

the inverted-V to represent the logical “and” operation, also called the “conjunction” operator

the rightward-pointing arrow to represent logical implication (that is, the “if-then” operation), and

the expression x-colon-t to represent the statement “x has type t”

When reading logical formulas aloud, the inverted-V can be read as the word “and”, and the arrow symbol can be read as the word “implies.”

CS 547 - Software Analysis

## From English to Inference Rule

- If  $e1$  has type  $int$  and  $e2$  has type  $int$ , then  $e1 + e2$  has type  $int$
- $(e1 \text{ has type } int \wedge e2 \text{ has type } int) \Rightarrow e1 + e2 \text{ has type } int$
- $(e1 : int \wedge e2 : int) \Rightarrow e1 + e2 : int$

Penn Engineering Property of Penn Engineering | 21

Let's see how we can express the following English sentence in our notation for rules of inference.

“If  $e1$  has type  $int$  and  $e2$  has type  $int$ , then  $e1+e2$  has type  $int$ .”

First, we can group the first two type declarations together with an inverted-V to represent their conjunction by the logical “and”.

Then we can replace the “if X then Y” construction with “X arrow Y,” where X is the conjoined hypotheses and Y is the conclusion “ $e1+e2$  has type  $int$ ”.

Finally, we can replace the phrases “X has type  $int$ ” with the notation “X colon  $int$ ”, giving us the following inference rule:

“ $e1$  has type  $int$  and  $e2$  has type  $int$  implies  $e1$  plus  $e2$  has type  $int$ ”

CS 547 - Software Analysis

## From English to Inference Rule

The statement

$$(e1 : \text{int} \wedge e2 : \text{int}) \Rightarrow e1 + e2 : \text{int}$$

is a special case of

$$\text{Hypothesis}_1 \wedge \dots \wedge \text{Hypothesis}_N \Rightarrow \text{Conclusion}$$

Penn Engineering Property of Penn Engineering | 22

Observe that in this formulation of the statement, we have two separate hypotheses which have been conjoined to form a compound hypothesis. Each separate hypothesis must be true in order to guarantee that the conclusion is true.

In general, inference rules have the form

Hypothesis 1 and Hypothesis 2 and so-forth upto Hypothesis N implies Conclusion.

An inference rule may have an arbitrary (but finite) number of hypotheses that must be fulfilled to guarantee the conclusion.

CS 547 - Software Analysis

## Notation for Inference Rules

- By tradition, inference rules are written
 
$$\frac{|- \text{Hypothesis}_1 \quad \dots \quad |- \text{Hypothesis}_N}{|- \text{Conclusion}}$$
- Hypotheses and conclusion are **type judgments**:
 
$$|- e : t$$
- $|-$  means “it is provable that...”

Penn Engineering Property of Penn Engineering | 23

Traditionally, inference rules are written in the following way.

Each inference rule has one conclusion and zero, one, or more hypotheses. The hypotheses are considered to be conjoined (“and”-ed) together: they must all be true in order to derive the conclusion. Inference rules without any hypotheses are called “axioms”: their conclusions are statements we take to be fact without any preconditions.

The hypotheses and conclusions all take the following form [point to “ $|- e : t$ ”] where  $e$  is an expression and  $t$  is a type. This statement is called a “type judgment.” The pipe symbol followed by a hyphen is a symbol denoting the phrase, “it is provable that.”

CS 547 - Software Analysis

## Rules for Integers

$$\frac{}{|- i : \text{int}} \quad [\text{Int}]$$

$$\frac{|- e_1 : \text{int} \quad |- e_2 : \text{int}}{|- e_1 + e_2 : \text{int}} \quad [\text{Add}]$$

Penn Engineering Property of Penn Engineering | 24

Let's start building the type system for our example language by focusing on the rules for integers.

The first rule we establish is that it is provable that  $i$  has type `int` for all literal integers  $i$ . For example we could replace the letter  $i$  with `0`, `1`, `200`, `-3`, etc. and the rule would still hold. Strictly speaking, we refer to such a template rule as a schema. And because there are no hypotheses in this rule, it is an axiom schema. We will name this schema `[Int]` for future reference.

The second rule we will establish is that if it is provable that  $e_1$  has type `int` and that  $e_2$  has type `int`, then it is provable that  $e_1 + e_2$  has type `int`. This is an inference rule schema (as opposed to an axiom schema) because there are more than 0 hypotheses. We will name this schema `[Add]`.

CS 547 - Software Analysis

## 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**

Penn Engineering Property of Penn Engineering | 25

These rules thus serve as templates describing how to type integers and expressions containing the addition operator. By filling in or instantiating the templates, we can produce complete typings for expressions.

Note that while hypotheses state facts about subexpressions, conclusions state facts about the entire expression.

CS 547 - Software Analysis

### Example: 1 + 2

$$\frac{\frac{}{|- 1 : \text{int}} \text{ [Int]} \quad \frac{}{|- 2 : \text{int}} \text{ [Int]}}{|- 1 + 2 : \text{int}} \text{ [Add]}$$

Penn Engineering Property of Penn Engineering | 26

Here's an example of a type derivation for typing the expression 1 + 2.

We first instantiate the axiom schema [Int], filling in the number 1 for i, to produce the axiom "it is provable that 1 has type int." Similarly, we instantiate the [Int] axiom schema again to produce the axiom "it is provable that 2 has type int."

Then, we can take these two conclusions and "plug them in" as hypotheses for the inference rule schema [Add]. Because it is provable that 1 has type int and 2 has type int, it is provable that 1+2 has type int. This conclusion completes the derivation of the typing for the expression 1+2.

CS 547 - Software Analysis

## SEGMENT

### Typing Rules: Part II

Penn Engineering Property of Penn Engineering | 27

CIS 547 - Software Analysis

## A Problem

What is the type of a variable reference?

$$\frac{|- e : \text{int}}{|- e + e : \text{int}}$$

Carries type information for **e** in hypotheses

$$\frac{}{|- x : ?} \quad [\text{Var}]$$

Doesn't carry enough information to give **x** a type

Penn Engineering Property of Penn Engineering | 28

We've discussed typing for integer expressions (which, in our simple language, are restricted to sums). Using the rules we've established so far, we can derive a rule such as this one: if it is provable that  $e$  has type  $\text{int}$ , then it is provable that  $e+e$  has type  $\text{int}$ . This schema carries the type information for  $e$  in its hypotheses.

Now, we also want to apply typing rules for variables. However, if we try to axiomatically define  $x$ 's type, we find that the axiom has no way of carrying the type information for  $x$  (and we don't want to impose a rule that all variables must be of a single type). Moreover, we can't provide type information via hypotheses, because these hypotheses must eventually be derived from axiomatic statements. So we've run into a problem. How can we provide appropriate type information for a variable?

CIS 547 - Software Analysis

## A Solution

- Put more information in the rules!
- An **environment** gives types for **free variables**

$$\frac{}{|- x : ?} \quad [\text{Var}]$$

  - 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

Penn Engineering Property of Penn Engineering | 29

The solution for our type system is to add more information to the rules.

We can use an environment to give type information to free variables, such as  $x$  in the rule  $[\text{Var}]$ .

A variable is free in an expression if it is not defined within the expression itself. In contrast, a variable that is defined in an expression is called a bound variable. For instance, consider the expression "lambda  $x:\text{int} \Rightarrow (x + y)$ " **[handwrite this]**. Variable  $y$  is free in this expression but variable  $x$  is bound.

In a type system, an environment is a function which maps variables to types. (For example, if we declare the variable  $x$  to be an  $\text{int}$  in a Java program, the environment would map the variable  $x$  to the type  $\text{int}$ .)

The concept of an environment is general: in other kinds of static analyses, environments may map variables to other abstract values instead of types.

## Type Environments

- Let  $A$  be a function from **variables** to **types**
- The sentence  $A \mid- e : t$  means:

“Under the assumption that variables have types given by  $A$ , it is provable that expression  $e$  has type  $t$ .”

Let's use the letter  $A$  to represent our environment; that is, our function from variables to types.

We'll use the notation “ $A$  pipe hyphen  $e$  colon  $t$ ” to mean: “under the assumption that variables have the types given by  $A$ , it is provable that expression  $e$  has type  $t$ ”.

## Modified Rules

- The type environment is added to all rules:

$$\begin{array}{c}
 A \mid- i : \text{int} \quad [\text{Int}] \\
 \\
 \frac{A \mid- e_1 : \text{int} \quad A \mid- e_2 : \text{int}}{A \mid- e_1 + e_2 : \text{int}} \quad [\text{Add}]
 \end{array}$$

Let's modify our previous rules [Int] and [Add] to include the type environment before the pipe-hyphen symbol.



CS 547 - Software Analysis

## A New Rule

- And we can write new rules:

$$\frac{}{A \vdash x : A(x)} \quad [\text{Var}]$$

Penn Engineering Property of Penn Engineering | 32

And now we can write new rules, such as this axiom schema for typing variables, which we call schema [Var].

“Under environment A, it is provable that x has type A(x)”. (A(x) refers to the type assigned to x under the environment mapping A.)

CS 547 - Software Analysis

## Rules for Functions

$$\frac{A[x \mapsto t] \vdash e : t'}{A \vdash \lambda x:t \Rightarrow e : t \rightarrow t'} \quad [\text{Def}]$$

A[x↦t] means “A modified to map x to type t”

$$\frac{A \vdash e1 : t1 \rightarrow t2 \quad A \vdash e2 : t1}{A \vdash e1 \ e2 : t2} \quad [\text{Call}]$$

Penn Engineering Property of Penn Engineering | 33

Now we can make the last of the inference rule schemas for our language. We need to introduce rules for typing function definitions and function calls.

For function definitions, we make the following rule, which we call [Def]:

Using our environment A, if it is provable that e is of type t-prime, then it is provable that the function, lambda x of type t yields e, is a function of type (t arrow t-prime).

Except, we have a problem. In this expression, x is a free variable, and we need its type. We can fix this by explicitly requiring the environment to map x to type t. We do so by adding the notation x arrow t in brackets beside the letter A in the hypothesis of this rule. Notice that this environment is the same as that in the conclusion of the rule, except that the environment in the conclusion of the rule does not give the type of x. This is correct since x is not a free variable in this expression: it is bound, or defined as the argument of this function.

Finally, we define the inference rule schema [Call] to give typing information for applications of functions to other expressions.

If, under the environment A, it is provable that e1 has type (t1 arrow t2) and e2 has type t1, then it is provable that the result of calling e1 with argument e2 has type t2.

CS 547 - Software Analysis

### All Rules Together

$$\frac{}{A \vdash i : \text{int}} \quad [\text{Int}]$$

$$\frac{A [x:t] \vdash e : t'}{A \vdash \lambda x:t \Rightarrow e : t \rightarrow t'} \quad [\text{Def}]$$

$$\frac{A \vdash e1 : \text{int} \quad A \vdash e2 : \text{int}}{A \vdash e1 + e2 : \text{int}} \quad [\text{Add}]$$

$$\frac{}{A \vdash x : A(x)} \quad [\text{Var}]$$

$$\frac{A \vdash e1 : t1 \rightarrow t2 \quad A \vdash e2 : t1}{A \vdash e1 e2 : t2} \quad [\text{Call}]$$

Penn Engineering Property of Penn Engineering | 34

This concludes the set of schemas we need in order to completely type our example lambda-calculus language. Notice that there is exactly one schema for each of the five syntactic expression forms in this language.

Now let's look at how we would derive the typing for a nontrivial expression using these five schemas. Later, we will see how to algorithmically compute these derivations, but let's work through some by hand first to get a feel for them.

CS 547 - Software Analysis

### Type Derivations: Example

$$\frac{\frac{\frac{}{[x:\text{int}] \vdash x : \text{int}}{[x:\text{int}] \vdash x + 1 : \text{int}}{[x:\text{int}] \vdash \lambda x:\text{int} \Rightarrow (x + 1) : \text{int} \rightarrow \text{int}}{[] \vdash (\lambda x:\text{int} \Rightarrow (x + 1)) (42) : \text{int}}}{[] \vdash \lambda x:\text{int} \Rightarrow (x + 1) : \text{int} \rightarrow \text{int}} \quad [\text{Def}] \quad \frac{}{[] \vdash 42 : \text{int}} \quad [\text{Int}]}{[] \vdash (\lambda x:\text{int} \Rightarrow (x + 1)) (42) : \text{int}} \quad [\text{Call}]$$

Penn Engineering Property of Penn Engineering | 35

A type derivation is a procedure which allows us to derive the type of a program, if one exists. The goal of the derivation is to show that an expression has a given type derived, ultimately, from axiomatic statements (that is, there are no infinite or cyclic chains of hypotheses needed to show that some statement is true).

In this slide we will use our five schemas to derive the type of the program “lambda x of type int yields x+1, applied to 42”. (See the instructor notes for a list of all the schemas for this language.)

We will work from the bottom up, checking to ensure that each single derivation fits the template for one of the rules we previously defined.

The [Call] rule schema requires that if applying expression e1 to expression e2 yields type t2, then expression e2 must be of type t1, and expression e1 must be a function mapping t1 to t2. In this case, the lambda expression plays the role of expression e1, the number 42 plays the role of expression e2, and t1 and t2 are both the int type. The [Call] rule schema is therefore satisfied.

Notice that we maintain an empty environment for this template: there are no variables mapped to types. We use an empty array to represent this environment with no such mappings.

Now let's derive the types of the hypotheses of the [Call] rule. The [Int] axiom

schema allows us to assert that the literal 42 has type int without requiring any further hypotheses.

If we look at the first hypothesis of the [Call] rule, we see we have a function definition. We therefore check to see that the [Def] rule schema can be applied to this definition. This schema requires that if the conclusion is a function mapping a variable x of type t1 to an expression of type t2, then the hypothesis must be that it is provable that the output expression is of type t2 under the same environment with a single additional mapping of the variable x to type t1.

In this case, notice that the environment in the conclusion is empty and the environment in the hypothesis maps the variable x to the type int. Under this assumption, the hypothesis states that it is provable x+1 (the expression the function outputs) is an int. Therefore, the [Def] rule schema is correctly instantiated, and we now proceed to prove that x+1 has type int under the environment where x is mapped to type int.

x+1 is an addition expression, so we now check that the [Add] rule schema holds. [Add] requires that the hypotheses and conclusion have the same environment, that the expression in the conclusion have type int, and that it be provable the expressions in the hypotheses also have type int. The schema has been correctly filled in, so now we must verify that each of the hypotheses are provable.

Like before, we have the [Int] axiom schema, under which the literal 1 is provable to be an int regardless of the environment. It is not a problem that x is in the environment but doesn't appear in the expression; we may still conclude that 1 is provably of type int.

We therefore focus on the last unproven statement: it is provable that x is of type int under the environment where x is mapped to type int. However, we have a different axiom schema, [Var], under which we may prove that a variable has type t under the environment where that variable is mapped to type t. Because the [Var] axiom schema may be instantiated by filling in t by the type int, the statement that x has type int is provable.

We are now done: all subexpressions of the entire program have been shown to be provable from axioms, leaving no unproven hypotheses. Therefore, we have completed the derivation, and proven that the full program has type int.

<#>

**QUIZ: Type Derivations**

[x:int, y:int] |- [ ] : [ ]

[x:int, y:int] |- [ ] : [ ]

---

[x:int, y:int] |- [ ] : [ ]

[x:int] |- λ y:int => (x + y) : [ ]

---

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

CIS 547 - Software Analysis Penn Engineering Property of Penn Engineering | 36

**{QUIZ SLIDE}**

Now let's do a quiz to let you practice a type derivation on your own.

Recall that in the previous quiz, you answered that the program “lambda x of type int yields the function (lambda y of type int yields x plus y)” has the type of a function mapping ints to (functions mapping ints to ints).

Let's suppose we wish to prove this type judgment under an empty environment. We take this to be the conclusion of a single derivation with the following hypothesis: “Under the environment mapping x to type int, it is provable that the function, lambda y of type int yields x plus y, has type BLANK.”

This hypothesis is the conclusion of another derivation whose hypothesis is: “Under the environment mapping x to type int and y to type int, it is provable that BLANK has type BLANK.”

This statement is the conclusion of yet another derivation with two hypotheses: “Under the environment mapping x to type int and y to type int, it is provable that BLANK has type BLANK,” and “Under the environment mapping x to type int and y to type int, it is provable that BLANK has type BLANK.”

The two hypotheses in this topmost rule are derived from the application of axioms

which are not shown for brevity.

Complete the type derivation by filling in each of the 7 blanks with appropriate expressions and types. If you need them as a reference, the five typing schemas for this language are shown in the instructor notes.

<#>

CIS 547 - Software Analysis

### QUIZ: Type Derivations

$[x:\text{int}, y:\text{int}] \vdash$   :

$[x:\text{int}, y:\text{int}] \vdash$   :

---

$[x:\text{int}, y:\text{int}] \vdash$   :

---

$[x:\text{int}] \vdash \lambda y:\text{int} \Rightarrow (x + y) :$

---

$[] \vdash \lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y)) :$

Penn Engineering Property of Penn Engineering | 37

#### {SOLUTION SLIDE}

Let's again work from the bottom up. We need to apply the [Def] inference rule schema to the last step in the derivation. This means we need the mapping  $x$  to  $\text{int}$  in the environment of the hypothesis, and we now need to prove that the type of the output expression is  $\text{int} \rightarrow \text{int}$ .

The expression,  $\lambda y$  of type  $\text{int}$  yields  $x + y$ , is now the conclusion of another derivation, which is another instance of the [Def] inference rule schema. We add the mapping  $y$  to  $\text{int}$  to the environment in the hypothesis, and it is now required to prove that the expression  $x+y$  has type  $\text{int}$ .

Finally, to prove that  $x+y$  has type  $\text{int}$ , we need to prove that under the environment  $x$  maps to  $\text{int}$  and  $y$  maps to  $\text{int}$ ,  $x$  has type  $\text{int}$  and  $y$  has type  $\text{int}$ . This is an application of the [Add] inference rule schema.

This completes the exercise, but note that, to complete this derivation, we would then need to use the [Var] axiom schema twice, once to derive that  $x$  has type  $\text{int}$  and again to derive that  $y$  has type  $\text{int}$ .

## READING

### From Javascript to Typescript

## LESSON

### Properties of Type Systems

CIS 547 - Software Analysis

## SEGMENT

# Type Soundness

Penn Engineering Property of Penn Engineering | 40

CIS 547 - Software Analysis

## Back to the Original Example

```

1: class T {
2:   int f(float a, int b,
3:     int[] c) {
4:     if (a)
5:       return b;
6:     else
7:       return c;
8:   }
9: }

```

File T.java

```

prompt$ javac T.java
T.java:4: error: incompatible types
    if (a)
       ^
    required: boolean
    found:    float
T.java:7: error: incompatible types
    return c;
           ^
    required: int
    found:    int[]
2 errors

```

Penn Engineering Property of Penn Engineering | 41

We are now done with the notation for type systems. Next, we will see some properties of type systems, such as:

What guarantees we have if a program type-checks (that is, it is successfully typed according to the axiom and inference rule schemas) versus if the program does not. And an algorithm to mechanically compute a type derivation, if it exists.

Let's return now to our Java example from the beginning of the lesson. In this case, the function `f` checks whether the variable `a` evaluates to true; if so, then it returns `b`; otherwise, it returns `c`. Our lambda-calculus language from earlier currently does not have the capacity to directly express this behavior, so let us extend the language to allow if-then-else statements. We'll need to add a new inference rule schema to check that if-then-else statements are correctly typed.

CS 547 - Software Analysis

## A More Complex Rule

$$\begin{array}{l}
 A \mid- e_0 : \text{bool} \\
 A \mid- e_1 : t_1 \\
 A \mid- e_2 : t_2 \\
 t_1 = t_2
 \end{array}
 \quad \text{[If-Then-Else]}$$


---


$$A \mid- \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : t_1$$

We'll use this rule to illustrate several ideas . . .

Penn Engineering Property of Penn Engineering | 42

The rule we'll add, which we'll call [If-Then-Else], has four hypotheses. If under the environment  $A$  it is provable that  $e_0$  has type `bool`,  $e_1$  has type  $t_1$ ,  $e_2$  has type  $t_2$ , and  $t_1$  equals  $t_2$ , then it is provable that the expression "if  $e_0$  then  $e_1$  else  $e_2$ " has type  $t_1$ .

Note that we're implicitly introducing a new type to our language, the `bool` type. We won't formally add all the required schemas to type boolean expressions in our language, but you can try writing these rules yourself as an exercise. For the purposes of this lesson, we will assume boolean expressions to be fully typed going forward.

Also notice we have a new form of hypothesis: " $t_1$  equals  $t_2$ ," which we do not preface with the "is provable that" notation. This expression is called a "side-condition." It is an additional constraint that must be satisfied in order for the inference rule schema to be instantiated.

We'll use this rule several times throughout the remainder of the lesson to illustrate some key concepts.

CS 547 - Software Analysis

## Type Soundness

$$\begin{array}{l}
 A \mid- e_0 : \text{bool} \\
 A \mid- e_1 : t_1 \\
 A \mid- e_2 : t_2 \\
 t_1 = t_2
 \end{array}
 \quad \text{[If-Then-Else]}$$


---


$$A \mid- \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : t_1$$

A type system is **sound** iff  
whenever 1.  $A \mid- e : t$  and  
2. If  $A(x) = t'$ , then  $x$  has a value  $v'$  in  $t'$   
then  $e$  evaluates to a value  $v$  in  $t$

Penn Engineering Property of Penn Engineering | 43

If we can successfully type the if-then-else statement, then the first hypothesis in the rule guarantees that  $e_0$  will be a boolean; the latter hypotheses and side condition guarantee that  $e_1$  and  $e_2$  will be of the same type.


These sorts of guarantees are characteristic of sound type systems. Formally, we define a type system to be sound if and only if:

Whenever it is provable under an environment  $A$  that  $e$  has type  $t$ , and furthermore, whenever the environment  $A$  maps variable  $x$  to type  $t'$ , then  $x$  has a value in that type  $t'$ , then  $e$  always evaluates to some value  $v$  in  $t$ .

CIS 547 - Software Analysis

## Comments on Soundness

- Soundness is extremely useful
  - Program type-checks => no errors at runtime
  - **Verifies** absence of a class of errors
- This is a very strong guarantee
  - Verified property holds in all executions
  - **“Well-typed programs cannot go wrong”**



**Robin Milner** (1924-2010)  
Turing Award, 1991

Penn Engineering Property of Penn Engineering | 44

Sound analyses are extremely useful. If a program type-checks according to a sound type system, then no errors of the kind checked by the type system can arise in any execution of the program. We say that a sound analysis verifies the absence of a class of errors.

This is a very strong guarantee: the property verified holds in all executions of the program. This guarantee is encapsulated by Robin Milner’s classic slogan, “Well-typed programs cannot go wrong.”

CIS 547 - Software Analysis

## Comments on Soundness

- 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

Penn Engineering Property of Penn Engineering | 45

But soundness comes at a price. Because of the undecidability of static analysis, a sound analysis that always terminates must occasionally generate spurious errors or warnings -- that is, false positives -- due to the abstraction the analysis makes. This means that the sound analysis is incomplete.

An alternative approach is to use an unsound analysis, which allows us more control over the false positive rate. However, unsound analyses have the possibility of failing to detect errors -- that is, producing false negatives.

Type systems are sound analyses: while they may generate false positives on occasion, they will never produce a false negative. But most kinds of analyses for detecting bugs -- bug-finding analyses -- are not sound. You can learn more about these kinds of analyses for Java and C by following the links below:

[For Java: “Using static analysis to find bugs”  
[http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=4602670&tag=1](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4602670&tag=1)]

[For C: “A few billion lines of code later: using static analysis to find bugs in the real world” <http://dl.acm.org/citation.cfm?id=1646374>]



CS 547 - Software Analysis

## QUIZ: Properties of Type Systems

Check the below untypable programs that can “go wrong”:

42 (λ x:int => (x + 5))	<input type="checkbox"/>
(λ x:int => x) + 1	<input type="checkbox"/>
if (true) then 1 else ((λ x:int => x) + 1)	<input type="checkbox"/>
(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))	<input type="checkbox"/>

Penn Engineering
Property of Penn Engineering | 46

{QUIZ SLIDE}

We saw that type systems are sound in that well-typed programs do not go wrong in any execution. But type systems are incomplete, that is, untypeable programs do not necessarily go wrong in some execution.

Let’s wrap up our discussion of the soundness of type systems with a quiz that illustrates the incompleteness of the type system we designed for our simple language.

Consider the following four programs. All of these programs are untypeable according to the rules we’ve described in this lesson. However, not all of them correspond to programs that would exhibit an error on execution. Check the boxes corresponding to which of these programs can “go wrong”: that is, there is some execution on which an error would occur.

CS 547 - Software Analysis

## QUIZ: Properties of Type Systems

Check the below untypable programs that can “go wrong”:

42 (λ x:int => (x + 5))	<input checked="" type="checkbox"/>
(λ x:int => x) + 1	<input checked="" type="checkbox"/>
if (true) then 1 else ((λ x:int => x) + 1)	<input type="checkbox"/>
(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))	<input type="checkbox"/>

Penn Engineering
Property of Penn Engineering | 47

{SOLUTION SLIDE}

Let’s consider each program in turn.

The first program is untypable because no rule allows for calling an integer like a function. Moreover, on every run of the program, this call to the integer would be encountered during execution, and it is undefined what should happen upon such a call. Therefore, this program “goes wrong” on all runs.

The second program is untypable again because none of our schemas allows for the addition of a function to an int. And, on every run of the program, this ill-defined addition would be encountered. So this program is another that could---and indeed always would---“go wrong.”

The third program is also untypable because the else clause contains a function added to an int. However, notice that no execution of this program would ever encounter this ill-defined addition, as the boolean expression always evaluates to true. This shows that our type system is incomplete: even though the program would always evaluate to the int value 1 and never encounter the “bad” code, we still reject it as ill-typed.

## QUIZ: Properties of Type Systems

Check the below untypable programs that can “go wrong”:

42 ( $\lambda x:\text{int} \Rightarrow (x + 5)$ )	<input checked="" type="checkbox"/>
$(\lambda x:\text{int} \Rightarrow x) + 1$	<input checked="" type="checkbox"/>
<code>if (true) then 1 else ((<math>\lambda x:\text{int} \Rightarrow x</math>) + 1)</code>	<input type="checkbox"/>
<pre>(if (c != 0) then (<math>\lambda x:\text{int} \Rightarrow x</math>)   else (<math>\lambda x:\text{int} \rightarrow \text{int} \Rightarrow (x 1)</math>)) (if (c != 0) then 1   else (<math>\lambda z:\text{int} \Rightarrow z</math>))</pre>	<input type="checkbox"/>

### {SOLUTION SLIDE}

The fourth program is untypable in our system because we require “then” and “else” clauses of the same boolean expression to evaluate to the same type; however, the functions output by the first boolean expression have different domains:  $\text{int} \rightarrow \text{int}$  and  $(\text{int} \rightarrow \text{int}) \rightarrow \text{int}$ , respectively. Similarly, the outputs of the second boolean expression are an  $\text{int}$  and  $\text{int} \rightarrow \text{int}$ .

Nevertheless, even though this program isn’t typable, we can see that on every run of the program, no ill-defined behavior would be encountered. If  $c$  is nonzero, then the first boolean expression would output a function mapping ints to ints, the second boolean expression would output 1, and we would then have the application of the function to 1.

On the other hand, if  $c$  were to equal 0, the first boolean expression would output a function which takes a function mapping ints to ints, and returning an int, while the second boolean expression would output a function mapping ints to ints, and so the resulting function application would also be well-defined.

This gives us two examples -- two false positives -- where our type system is overly conservative about what it considers to be a valid program. While our type system can be proven to be sound, we will never be without these false positives, even if we try to introduce new rules allowing us to detect whether the “then” or “else” branches of

boolean expressions are followed.



## LESSON

Type Inference



## SEGMENT

Type Inference

CS 547 - Software Analysis

## Type Constraints

```

A |- e0 : bool
A |- e1 : t1
A |- e2 : t2
t1 = t2

```

Side constraints must be solved

---

```

A |- if e0 then e1 else e2 : t1

```

```

if (a > 1)
  then (λ x:int => x)
  else (10)

```

Many analyses have **side conditions**

- Often **constraints** to be solved
- All constraints must be **satisfied**
- A separate **algorithmic** problem

Penn Engineering Property of Penn Engineering | 51

Let's look some more at the notion of side conditions. These conditions occur in many type-based analyses. They are constraints that must be solved -- that is, satisfied -- in order to verify that the inference rule is properly applied. Checking whether these side constraints can be solved is a separate algorithmic problem that we will cover shortly.

For example, consider the program "if (a > 1) then (lambda x:int => x) else 10" in an environment where a is of type int. Expression e1 in this program has type int -> int, while expression e2 has type int. Without the side-condition, this program would be typed as a function mapping ints to ints; this would cause the type system to be unsound, as the program would not evaluate to the correct type if 'a' were ever less than or equal to 1.

CS 547 - Software Analysis

## Another Example

- Consider a **recursive function**

$$f(x) = \dots f(e) \dots$$
- If  $x : t1$  and  $e : t2$  then  $t2 = t1$ 
  - Can be relaxed to  $t2 \subseteq t1$
- Recursive functions yield **recursive constraints**
  - Same with **loops**
  - How hard constraints are to solve depends on constraint language, details of application

Penn Engineering Property of Penn Engineering | 52

Another example of where side conditions occur are with recursive functions: functions which call themselves in their own definition. The constraint that must be satisfied is that if x is of type t1 and e is of type t2, then t2 equals t1. (In fact, this is a bit too strict; it could be relaxed to requiring only that t2 is a subset of t1.)

Recursive functions yield constraints that are themselves recursive. The same holds for loops. We observed this phenomenon in the module on dataflow analysis, wherein loops in programs necessitate multiple iterations of the chaotic iteration algorithm. That algorithm in reality is solving a system of recursive constraints, namely, dataflow constraints for computing the IN sets and OUT sets of dataflow facts at each program point.

In general, the difficulty of solving constraints depends on the language of the constraints and the details of the application.

CS 547 - Software Analysis

## Type Inference

- Languages do not require types for entire programs to reduce verbosity
- **Type inference** determines the type of a program, if it has one
- Also called **type checking** or **type reconstruction** since some types are usually specified (e.g. variable declarations and function signatures)
- A challenging problem for complex type systems and tricky languages  
=> many tradeoffs (sound vs. complete vs. efficient vs. automatic ...)

Property of Penn Engineering | 53

Another important property of type systems is that they do not require types for entire programs, and for good reason: the goal is to alleviate the annotation burden on the programmer by reducing verbosity.

The problem of determining the type of the entire program, if it has one, is called type inference. Since some types are usually specified, in elements such as variable declarations and function signatures, type inference is also called type checking or type reconstruction.

Type inference is a challenging problem for complex type systems such as Haskell's, and for tricky languages, such as Javascript. Not surprisingly, akin to any software analysis, there are many tradeoffs that a type inference algorithm has to strike, in aspects such as soundness, completeness, efficiency, and automation.

CS 547 - Software Analysis

## Outline of Type Inference Algorithm

- **Step 1:** Generate constraints based on the type rules
- **Step 2:** Solve the constraints to obtain types or show program is untypable
  - **Unification algorithm** → automated/interactive theorem prover
- Introduce a type variable (denoted  $[[...]]$ ) for each program fragment
  - **Variable** – e.g.  $[[x]]$  and
  - **Expression** – e.g.  $[[e1]]$

Property of Penn Engineering | 54

Let's look at the outline of a type inference algorithm. The algorithm uses constraints to uniformly solve both the problems we just discussed: how to check side conditions arising in type rules and how to infer the type of the entire program, if one exists.

As we learnt in the module on constraint-based analysis, this algorithm involves two steps: first collect constraints based on the type rules. Once we have generated a set of constraints, we solve them using a simple but powerful algorithm called unification. Unification is highly efficient and runs in approximately linear time in the size of the input program.

If the constraints can be solved, the solution will provide types for each program fragment, which in the case of our lambda calculus language is each variable and each expression. If the constraints are unsatisfiable, there is a type error, and the program is not typeable.

For richer type systems, constraint-based type inference might use an off-the-shelf automated theorem prover or even an interactive theorem prover.

However, for this type system, we will illustrate the constraint-solving process using the unification algorithm through a series of examples. Throughout these examples we will denote "typeof" as double square brackets  $[[...]]$

**SEGMENT**

## Type Inference: Example 1 of 2

**Type Inference: Example 1**

```
λ x:int => (λ y:int => (x + y))
```

Suppose we would like to infer the type of the program shown here.

CIS 547 - Software Analysis

### Type Inference: Example 1

$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$

$A \mid - e1 : \text{int}$	$A \mid - e2 : \text{int}$	[Add]
$A \mid - e1 + e2 : \text{int}$		

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$

Penn Engineering Property of Penn Engineering | 57

First, we will generate constraints based on type rules.

Let's start out with the expression (x+y) highlighted here. From the type rule for addition, we know that the types of sub-expressions x and y must be int, and furthermore, that the type of the expression (x+y) must also be int.

CIS 547 - Software Analysis

### Type Inference: Example 1

$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$

$A [x:t] \mid - e : t'$	[Def]
$A \mid - \lambda x:t \Rightarrow e : t \rightarrow t'$	

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = [[y]] \Rightarrow [[x + y]]$

Penn Engineering Property of Penn Engineering | 58

Likewise, by the type rule for function definitions, we can conclude the type of the function highlighted in blue must be equal to (type of the parameter, y, arrow the type of the body, x+y).

CS 547 - Software Analysis

### Type Inference: Example 1

$$\frac{A [x:t] \vdash e : t'}{A \vdash \lambda x:t \Rightarrow e : t \rightarrow t'} \text{ [Def]}$$

$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = [[y]] \Rightarrow [[x + y]]$
5.  $[[\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))]] = [[x]] \Rightarrow [[\lambda y:\text{int} \Rightarrow (x + y)]]$

Now let's solve the constraints ...

Penn Engineering Property of Penn Engineering | 59

Lastly, we generate constraints for the type of the entire program. Again we use the type rule for function definitions.

Now that we have generated all the constraints, we proceed to Step 2: solving them using the unification algorithm.

CS 547 - Software Analysis

### Type Inference: Example 1

$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = [[y]] \Rightarrow [[x + y]]$
5.  $[[\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))]] = [[x]] \Rightarrow [[\lambda y:\text{int} \Rightarrow (x + y)]]$

Penn Engineering Property of Penn Engineering | 60

Informally, at each step of the unification algorithm, we simply pick a substitution from a constraint and apply it to all the remaining constraints. The goal is to attempt to “unify” the two terms in each constraint – that is, to make them identical – by applying a substitution.

For example, constraint 1, type of x equals int, would substitute int wherever type of x appears in the remaining constraints.

If a contradiction arises during this process, then the program cannot be typed. If no contradiction arises, then the substitution will have given us the type of each program part including the type of the entire program.

We will illustrate the unification algorithm by applying substitution  $[[x]] = \text{int}$  from constraint 1 to the remaining constraints. This simplifies constraint 5 to ...



## Type Inference: Example 1

$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = [[y]] \Rightarrow [[x + y]]$
5.  $[[\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))]] = \text{int} \Rightarrow [[\lambda y:\text{int} \Rightarrow (x + y)]]$

this.

## Type Inference: Example 1

$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = [[y]] \Rightarrow [[x + y]]$
5.  $[[\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))]] = \text{int} \Rightarrow [[\lambda y:\text{int} \Rightarrow (x + y)]]$

Marching ahead, we next apply the substitution from constraint 2 to all the remaining constraints.

$[[y]]$  only occurs in constraint 4. So, we substitute  $\text{int}$  for  $[[y]]$  in constraint 4 to obtain

...

## Type Inference: Example 1

$$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$$

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = \text{int} \Rightarrow [[x + y]]$
5.  $[[\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))]] = \text{int} \Rightarrow [[\lambda y:\text{int} \Rightarrow (x + y)]]$

this.

## Type Inference: Example 1

$$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$$

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = \text{int} \Rightarrow [[x + y]]$
5.  $[[\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))]] = \text{int} \Rightarrow [[\lambda y:\text{int} \Rightarrow (x + y)]]$

Next we apply the substitution from constraint 3.

Again,  $[[x+y]]$  only occurs in constraint 4 so we substitute it by `int` to obtain ...

## Type Inference: Example 1

$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = \text{int} \Rightarrow \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))]] = \text{int} \Rightarrow [[\lambda y:\text{int} \Rightarrow (x + y)]]$

this.

## Type Inference: Example 1

$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = \text{int} \Rightarrow \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))]] = \text{int} \Rightarrow [[\lambda y:\text{int} \Rightarrow (x + y)]]$

Now, we choose constraint 4 for substitution. The left hand side of constraint 4 occurs in constraint 5, so we substitute it with  $\text{int} \Rightarrow \text{int}$  to obtain ...

CIS 547 - Software Analysis

## Type Inference: Example 1

$\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))$

$\text{int} \Rightarrow (\text{int} \Rightarrow \text{int})$

1.  $[[x]] = \text{int}$
2.  $[[y]] = \text{int}$  ✓
3.  $[[x + y]] = \text{int}$
4.  $[[\lambda y:\text{int} \Rightarrow (x + y)]] = \text{int} \Rightarrow \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (\lambda y:\text{int} \Rightarrow (x + y))]] = \text{int} \Rightarrow (\text{int} \Rightarrow \text{int})$

Penn Engineering Property of Penn Engineering | 67

This substitution has given us the type of the entire program  $\text{int} \Rightarrow (\text{int} \Rightarrow \text{int})$ . It is a function which takes an `int` and returns a function that takes an `int` and returns an `int`. In summary, we were able to successfully solve the constraints and infer the type of each program part.

CIS 547 - Software Analysis

## SEGMENT

### Type Inference: Example 2 of 2

Penn Engineering Property of Penn Engineering | 68

CIS 547 - Software Analysis

## Type Inference: Example 2

$42 (\lambda x:\text{int} \Rightarrow (x + 5))$

Property of Penn Engineering | 69

Now let's look at an example where type inference fails, that is, the program is untypable. Consider the following program.

CIS 547 - Software Analysis

## Type Inference: Example 2

$42 (\lambda x:\text{int} \Rightarrow (x + 5))$

$$\frac{}{A \vdash i : \text{int}} \quad [\text{Int}]$$

1.  $[[42]] = \text{int}$

Property of Penn Engineering | 70

As before, we generate constraints for each program part using our previously defined type rules. Our type rule for integer constants gives us types for the constants 42 and 5.

CS 547 - Software Analysis

## Type Inference: Example 2

**42**  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$

$$\frac{}{A \vdash i : \text{int}} \quad [\text{Int}]$$

Property of Penn Engineering | 71

CS 547 - Software Analysis

## Type Inference: Example 2

**42**  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$

$$\frac{A \vdash e_1 : \text{int} \quad A \vdash e_2 : \text{int}}{A \vdash e_1 + e_2 : \text{int}} \quad [\text{Add}]$$

Property of Penn Engineering | 72

Now we generate the constraints for the expression  $(x+5)$ . From the type rule for addition, we conclude that the type of sub-expressions  $x$  and  $5$  must be `int`. We already have a constraint stating that the type of  $5$  must be `int`, so we only add the constraint stating that the type of  $x$  must be `int`. Furthermore, the type rule states that the type of the expression  $(x+5)$  must be `int`.

CIS 547 - Software Analysis

### Type Inference: Example 2

**42  $(\lambda x:\text{int} \Rightarrow (x + 5))$**

$$\frac{A [x:t] \vdash e : t'}{A \vdash \lambda x:t \Rightarrow e : t \rightarrow t'} \text{ [Def]}$$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = [[x]] \Rightarrow [[x + 5]]$

Penn Engineering
Property of Penn Engineering | 73

Next, we generate the constraints for the highlighted function. By the type rule for function definitions, we know that the type must be equal to a function type from the type of the parameter to the type of the body.

CIS 547 - Software Analysis

### Type Inference: Example 2

**42  $(\lambda x:\text{int} \Rightarrow (x + 5))$**

$$\frac{A \vdash e_1 : t_1 \rightarrow t_2 \quad A \vdash e_2 : t_1}{A \vdash e_1 e_2 : t_2} \text{ [Call]}$$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = [[x]] \Rightarrow [[x + 5]]$
6.  $[[42]] = [[\lambda x:\text{int} \Rightarrow (x + 5)]] \Rightarrow$   
 $[[42 (\lambda x:\text{int} \Rightarrow (x + 5))]]$

Now let's solve the constraints ...

Penn Engineering
Property of Penn Engineering | 74

Lastly, we use the type rule for function calls to obtain the constraint on the type of the entire program, namely, that the type of the called function  $[[42]]$  should be a function type from the type of the argument, which is itself a function, to the return result, which is the entire program.

Clearly, we can see that this cannot possibly be the case, since the type of 42 cannot simultaneously be an integer, as dictated by constraint 1, and a function, as dictated by constraint 6.

Nevertheless, let's continue with the unification algorithm to see where this contradiction arises.

## Type Inference: Example 2

42  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = [[x]] \Rightarrow [[x + 5]]$
6.  $[[42]] = [[\lambda x:\text{int} \Rightarrow (x + 5)]] \Rightarrow$   
 $[[42 (\lambda x:\text{int} \Rightarrow (x + 5))]]$

First, we select constraint 1 and substitute int for [[42]] in constraint 6 to obtain...

## Type Inference: Example 2

42  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = [[x]] \Rightarrow [[x + 5]]$
6.  $\text{int} = [[\lambda x:\text{int} \Rightarrow (x + 5)]] \Rightarrow$   
 $[[42 (\lambda x:\text{int} \Rightarrow (x + 5))]]$

this.



CIS 547 - Software Analysis

### Type Inference: Example 2

**42**  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = [[x]] \Rightarrow [[x + 5]]$
6.  $\text{int} = [[\lambda x:\text{int} \Rightarrow (x + 5)]] \Rightarrow$   
 $[[42 (\lambda x:\text{int} \Rightarrow (x + 5))]]$

Penn Engineering Property of Penn Engineering | 77

Similarly, we select constraint 3 for substitution in constraint 5 to obtain ...

CIS 547 - Software Analysis

### Type Inference: Example 2

**42**  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = \text{int} \Rightarrow [[x + 5]]$
6.  $\text{int} = [[\lambda x:\text{int} \Rightarrow (x + 5)]] \Rightarrow$   
 $[[42 (\lambda x:\text{int} \Rightarrow (x + 5))]]$

Penn Engineering Property of Penn Engineering | 78

this.

CS 547 - Software Analysis

### Type Inference: Example 2

**42**  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = \text{int} \Rightarrow [[x + 5]]$
6.  $\text{int} = [[\lambda x:\text{int} \Rightarrow (x + 5)]] \Rightarrow$   
 $[[42 (\lambda x:\text{int} \Rightarrow (x + 5))]]$

Penn Engineering Property of Penn Engineering | 79

Next, we select constraint 4 for substitution.  $[[x+5]]$  only exists in constraint 5, so we perform the substitution to int there.

CS 547 - Software Analysis

### Type Inference: Example 2

**42**  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = \text{int} \Rightarrow \text{int}$
6.  $\text{int} = [[\lambda x:\text{int} \Rightarrow (x + 5)]] \Rightarrow$   
 $[[42 (\lambda x:\text{int} \Rightarrow (x + 5))]]$

Penn Engineering Property of Penn Engineering | 80

CS 547 - Software Analysis

### Type Inference: Example 2

**42**  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = \text{int} \Rightarrow \text{int}$
6.  $\text{int} = [[\lambda x:\text{int} \Rightarrow (x + 5)]] \Rightarrow [[42 (\lambda x:\text{int} \Rightarrow (x + 5))]]$

Penn Engineering Property of Penn Engineering | 81

Lastly, we use constraint 5 to substitute the type of the highlighted function in constraint 6.

CS 547 - Software Analysis

### Type Inference: Example 2

**42**  $(\lambda x:\text{int} \Rightarrow (x + 5))$

1.  $[[42]] = \text{int}$
2.  $[[5]] = \text{int}$
3.  $[[x]] = \text{int}$
4.  $[[x + 5]] = \text{int}$
5.  $[[\lambda x:\text{int} \Rightarrow (x + 5)]] = \text{int} \Rightarrow \text{int}$
6.  $\text{int} = \text{int} \Rightarrow \text{int} \Rightarrow [[42 (\lambda x:\text{int} \Rightarrow (x + 5))]]$

$\text{int} \neq (\text{int} \Rightarrow \text{int}) \Rightarrow \alpha$

**X**

Penn Engineering Property of Penn Engineering | 82

Here, a contradiction arises. It cannot be possible for int to equal a function type. Thus, unification has failed, and this program is untypable, as expected.



## READING

### A Formal Type Inference Algorithm



## LESSON

### Notation for Static Analyses

CS 547 - Software Analysis

## SEGMENT

# Static Analyses Using Type Rules

Penn Engineering

Property of Penn Engineering | 85

CS 547 - Software Analysis

## Static Analyses Using Type Rules

$$\frac{A1 \mid - e1 : t1 \quad A2 \mid - e2 : t2}{A \mid - e : t}$$

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  $e$

Analysis of expression is recursively defined using analysis of subexpressions

Penn Engineering

Property of Penn Engineering | 86

Earlier in this module, we promised that type notation would be useful in contexts aside from type systems. Indeed, they are useful for describing a wide range of static analyses. Instead of speaking in the language of types, we can interpret each part of a rule of inference in a different way:

We will use  $e$  to represent the program (or program fragment) to be analyzed.

$t$  will represent an abstract value (which need not necessarily be a type) computed for the program  $e$ .

Instead of mapping variables to types, the environment  $A$  may now describe assumptions needed for aspects of  $e$  that are determined by the rest of the program.

This will allow us to recursively analyze hypotheses of each inference rule -- subexpressions of the program -- to prove that the program has some particular global property.

What sort of properties can we reason about in this manner?

## An Example: The Rule of Signs

- Goal: to estimate the sign of a numeric computation
- Example:  $-3 * 4 = -12$
- Abstraction:  $- * + = -$

As a starting example, let's use type notation to determine the sign of a numeric computation. For example, we would like to prove that the output of -3 times 4 is negative. Abstracting this particular computation, we can represent it as "negative times positive equals negative."

## Abstract Values

- $+$  = { all positive integers }
- $0$  = { 0 }
- $-$  = { all negative integers }
- Environment  $A$  : Variables  $\rightarrow$  { +, 0, - }

The abstract values that we will use in this analysis are:

PLUS, representing the set of all positive integers;  
ZERO, representing the set consisting of just the integer 0; and  
MINUS, representing the set of all negative integers.

The environment in this system will be a mapping from variables to one of these three abstract values.

CS 547 - Software Analysis

### QUIZ: Example Rules – Part I

Fill in the boxes with +, -, or 0 as appropriate.

$\frac{A \mid - e1 : + \quad A \mid - e2 : -}{A \mid - e1 * e2 : \quad}$	$\frac{A \mid - e1 : + \quad A \mid - e2 : +}{A \mid - e1 * e2 : \quad}$
$\frac{A \mid - e1 : - \quad A \mid - e2 : -}{A \mid - e1 * e2 : \quad}$	$\frac{A \mid - e1 : 0 \quad A \mid - e2 : +}{A \mid - e1 * e2 : \quad}$

Penn Engineering Property of Penn Engineering | 89

{QUIZ SLIDE}

Let's do a quiz on writing some example rules of inference that we would want to add to our analysis.

Under the environment A, if it is provable that e1 has value PLUS and e2 has value MINUS, then it is provable under A that e1 times e2 has value BLANK.

Under the environment A, if it is provable that e1 has value PLUS and e2 has value PLUS, then it is provable under A that e1 times e2 has value BLANK.

Under the environment A, if it is provable that e1 has value MINUS and e2 has value MINUS, then it is provable under A that e1 times e2 has value BLANK.

Under the environment A, if it is provable that e1 has value ZERO and e2 has value PLUS, then it is provable under A that e1 times e2 has value BLANK.

As a quick check of your understanding, enter the appropriate abstract value for the conclusion of each inference rule. Enter either the plus symbol, the minus symbol, or the number 0.

CS 547 - Software Analysis

### QUIZ: Example Rules – Part I

Fill in the boxes with +, -, or 0 as appropriate.

$\frac{A \mid - e1 : + \quad A \mid - e2 : -}{A \mid - e1 * e2 : \quad -}$	$\frac{A \mid - e1 : + \quad A \mid - e2 : +}{A \mid - e1 * e2 : \quad +}$
$\frac{A \mid - e1 : - \quad A \mid - e2 : -}{A \mid - e1 * e2 : \quad +}$	$\frac{A \mid - e1 : 0 \quad A \mid - e2 : +}{A \mid - e1 * e2 : \quad 0}$

Penn Engineering Property of Penn Engineering | 90

{SOLUTION SLIDE}

Recall that a positive number times a negative number should be a negative number, so:

Under the environment A, if it is provable that e1 has value PLUS and e2 has value MINUS, then it is provable under A that e1 times e2 has value MINUS.

Further, a positive number times a positive number should be positive, so:

Under the environment A, if it is provable that e1 has value PLUS and e2 has value PLUS, then it is provable under A that e1 times e2 has value PLUS.

Similarly, a negative times a negative is positive, so:

Under the environment A, if it is provable that e1 has value MINUS and e2 has value MINUS, then it is provable under A that e1 times e2 has value PLUS.

Finally, zero times any number is zero, so:

Under the environment A, if it is provable that e1 has value ZERO and e2 has value PLUS, then it is provable under A that e1 times e2 has value ZERO.

Note that these don't constitute all possible rules we would want to add: we would need to add more to fully type all possible expressions (not to mention that we would need to add axioms like [Int] and [Var] from our lambda-calculus language).

CIS 547 - Software Analysis

**SEGMENT**

Revisiting Abstract Domains

Penn Engineering Property of Penn Engineering | 91

CIS 547 - Software Analysis

**Revisiting Abstract Domains**

$$\frac{A \mid - e1 : + \quad A \mid - e2 : -}{A \mid - e1 + e2 : ?}$$

We don't have an abstract value that covers this case!

**Solution:**

Add abstract values to ensure **closure** under all operations:

- + = { all positive integers }
- 0 = { 0 }
- = { all negative integers }
- TOP = { all integers }
- BOT = { }

Penn Engineering Property of Penn Engineering | 92

So far, we have described rules for deriving the sign of the product of two expressions. However, what if we want to describe the sign of the sum of two expressions? Let's add the following rule:

Under environment A, if it is provable that e1 has value PLUS and e2 has value MINUS, then it is provable under A that e1 plus e2 has value ... what?

We have a problem here. We don't have an abstract value that covers this case.

How do we solve this problem? The solution is to add new abstract values to our system to make sure the system is closed under all operations: there are no expressions with undefined abstract values.

To our existing abstract values, we will add two additional abstract values:

TOP, which will represent the set of all integers

BOT, which will represent the empty set

Notice that the abstract values no longer represent disjoint sets of concrete values: this is OK! This happens quite regularly in abstract domains, as we saw in the module on dataflow analysis, and earlier in the module on introduction to software analysis.



CIS 547 - Software Analysis

## QUIZ: Example Rules – Part II

Fill in the boxes with +, -, 0, TOP, or BOT as appropriate.

$\frac{A \mid - e1 : + \quad A \mid - e2 : -}{A \mid - e1 + e2 : \boxed{\phantom{000}}}$	$\frac{A \mid - e1 : + \quad A \mid - e2 : +}{A \mid - e1 + e2 : \boxed{\phantom{000}}}$
$\frac{A \mid - e1 : 0 \quad A \mid - e2 : +}{A \mid - e1 / e2 : \boxed{\phantom{000}}}$	$\frac{A \mid - e1 : TOP \quad A \mid - e2 : 0}{A \mid - e1 / e2 : \boxed{\phantom{000}}}$

Penn Engineering Property of Penn Engineering | 93

**{QUIZ SLIDE}**

For this quiz, let's try to determine the appropriate abstract value for the conclusion in each of the following inference rules:

Under the environment A, if it is provable that e1 has value PLUS and e2 has value MINUS, then it is provable under A that e1 plus e2 has value BLANK.

Under the environment A, if it is provable that e1 has value PLUS and e2 has value PLUS, then it is provable under A that e1 plus e2 has value BLANK.

Under the environment A, if it is provable that e1 has value ZERO and e2 has value PLUS, then it is provable under A that e1 divided by e2 has value BLANK.

Under the environment A, if it is provable that e1 has value TOP and e2 has value ZERO, then it is provable under A that e1 divided by e2 has value BLANK.

Notice that we're also introducing rules of inference about division now: just use standard rules of arithmetic to determine your answers here.

Fill in each of the boxes with a plus, minus, zero, the word TOP, or the word BOT as appropriate.

CIS 547 - Software Analysis

## QUIZ: Example Rules – Part II

Fill in the boxes with +, -, 0, TOP, or BOT as appropriate.

$\frac{A \mid - e1 : + \quad A \mid - e2 : -}{A \mid - e1 + e2 : \boxed{TOP}}$	$\frac{A \mid - e1 : + \quad A \mid - e2 : +}{A \mid - e1 + e2 : \boxed{+}}$
$\frac{A \mid - e1 : 0 \quad A \mid - e2 : +}{A \mid - e1 / e2 : \boxed{0}}$	$\frac{A \mid - e1 : TOP \quad A \mid - e2 : 0}{A \mid - e1 / e2 : \boxed{BOT}}$

Penn Engineering Property of Penn Engineering | 94

**{SOLUTION SLIDE}**

Let's start with the problematic example we considered earlier: if we add a positive number to a negative number, we could end up with a positive, negative, or zero sum. Therefore,

Under the environment A, if it is provable that e1 has value PLUS and e2 has value MINUS, then it is provable under A that e1 plus e2 has value TOP.

However, if we add two positive numbers, the outcome will definitely be a positive number. Therefore,

Under the environment A, if it is provable that e1 has value PLUS and e2 has value PLUS, then it is provable under A that e1 plus e2 has value PLUS.

Next, we consider division of zero by a positive number. This will definitely always output zero, so

Under the environment A, if it is provable that e1 has value ZERO and e2 has value PLUS, then it is provable under A that e1 divided by e2 has value ZERO.

On the other hand, division of any number by zero is undefined, so

Under the environment A, if it is provable that e1 has value TOP and e2 has value ZERO, then it is provable under A that e1 divided by e2 has value BOT.

These rules would, of course, need to be augmented with several more to express the

entire sign computation analysis. As an exercise, try to write down the rules to extend this system to the four basic arithmetic operations: addition, subtraction, multiplication, and division. You can even try to add the modulus operation if you like!

CIS 547 - Software Analysis

# SEGMENT

## Flow and Path Sensitivity

Penn Engineering

Property of Penn Engineering | 95

CS 547 - Software Analysis

## Flow Insensitivity

```

A |- e0 : bool
A |- e1 : t1
A |- e2 : t2
t1 = t2
-----
A |- if e0 then e1 else e2 : t1

```

Subexpressions are independent of each other

- **Flow-insensitive analysis:** analysis is independent of ordering of sub-expressions
- => analysis result unaffected by permuting statements
- **Type systems** are generally flow-insensitive

Penn Engineering Property of Penn Engineering | 96

Backtracking a bit, let's return to our analysis of the if-then-else expression from our lambda-calculus language. (It may seem like a digression from our current topic, but it will wrap back up nicely.)

Recall that in our algorithm's examination of the if-then-else expression, each of the subexpressions were analyzed independently of one another. It didn't take into account the order in which the expressions were listed; it only checked that  $e_0$ ,  $e_1$ , and  $e_2$  were of the appropriate types. We could have swapped  $e_1$  and  $e_2$ , and the analysis would be unaffected.

This is called flow-insensitive analysis, which you may recall from our study of pointer analysis. The analysis is independent of the order in which the statements appear in the program; in effect, it treats the program as a set of statements with little other structure imposed.

In practice, you'll find that most type systems are flow-insensitive.

CS 547 - Software Analysis

## Comments on Flow Insensitivity

- Flow insensitive analyses are often very **efficient and scalable**
- No need for modeling a separate state for each subexpression
- But can be **imprecise** ...

Penn Engineering Property of Penn Engineering | 97

The reason for disregarding the control flow of the program is the usual reason for abstraction in static analysis: it simplifies the representation of the program state that our algorithm operates on. In particular, there is no need for modeling a separate state for each subexpression.

The resulting analyses are efficient and scalable, allowing them to be used liberally in many applications.

The tradeoff is loss of precision -- the usual tradeoff for speed in static analysis. To attain a faster, sound type system, we necessarily end up generating more false positives. (Indeed, we have a constant time type-checking algorithm that's perfectly sound: just reject all programs! But of course such a scheme, while fast and sound, is completely useless.)

CS 547 - Software Analysis

## Flow Sensitivity

```

A |- e0 : bool ▷ A0
A0 |- e1 : t1 ▷ A1
A0 |- e2 : t2 ▷ A2
t1 = t2, A1 = A2
-----
A |- if e0 then e1 else e2 : t1 ▷ A1

```

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 an example of a flow-sensitive analysis

Penn Engineering Property of Penn Engineering | 98

If we are willing to sacrifice simplicity and efficiency in order to increase our precision, we can make our abstraction of the program state more realistic. One way to do so is to remain sensitive to control flow.

In this case, each statement has an “input” environment and an “output” environment, and we order the analysis of subexpressions by the environments that are produced. Therefore, the result of the analysis now depends on the order of the statements, allowing us to make a finer distinction between correctly-typed and incorrectly-typed statements.

In the [If-Then-Else] schema example, we would start by proving that under environment A, e0 is a bool, and then output a new environment A0. This environment then flows into the analysis of both the statement that e1 is of type t1 and that e2 is of type t2, each of which produce their own “output” environments A1 and A2. We would then need to solve the side condition that environments A1 and A2 are equal (in addition to the previous side condition that t1 = t2) in order to prove that, under the environment A, the expression “if e0 then e1 else e2” has type t1 and output environment A1.

This kind of analysis is called flow-sensitive analysis: the analysis result depends on the order of statements in the program.

Now is where we start tying together our previous thread---on different types of static

analysis---with this thread. The “input” and “output” environments in this kind of static analysis should remind you of something we saw earlier: in dataflow analysis, we had “in” and “out” sets of program facts. These sets would play the part of the input and output environments. Therefore, dataflow analysis (as we might expect from its name) is a flow-sensitive static analysis which could be represented using type-system-like notation.

Comments on Flow Sensitivity

- Example:
  - Rule of signs extended with assignment statements
 
$$\frac{A \mid - e : + \triangleright A}{A \mid - x := e \triangleright A[x \mapsto +]}$$
  - $A[x \mapsto +]$  means  $A$  modified so that  $A(x) = +$
- Flow-sensitive analysis can be expensive
  - Each statement has own **model of state**
  - Polynomial** cost increase over flow-insensitive

Penn Engineering Property of Penn Engineering | 99

The environments shown previously in flow-sensitive analysis were fairly abstract. Let's look at a more concrete example by extending our rule-of-signs system with assignment statements.

We add the following inference rule schema:

Under environment  $A$ , if it is provable that  $e$  has value PLUS and results in environment  $A$ , then, under environment  $A$ , it is provable that  $x$  assigned the expression  $e$  results in the environment  $A$ , modified so that  $A$  maps  $x$  to the value PLUS.

As we alluded to earlier, flow-sensitive analysis can be expensive. Our abstraction is more complex: each statement has to maintain its own separate model of the program's state. Because we are keeping track of more information, we end up with a polynomial increase in cost over flow-insensitive analyses.

Path Sensitivity

Predicate is refined at decision points (e.g., if's)

$$\begin{array}{l} P, A \mid - e_0 : \text{bool} \triangleright A_0 \\ P \wedge e_0, A_0 \mid - e_1 : t_1 \triangleright A_1 \\ P \wedge \neg e_0, A_0 \mid - e_2 : t_2 \triangleright A_2 \\ t_1 = t_2 \\ \hline P, A \mid - \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : t_1, e_0 ? A_1 : A_2 \end{array}$$

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

Penn Engineering Property of Penn Engineering | 100

We can further refine our model of the program's state in order to improve the precision of our analysis. Recall that, for the statement "if  $e_0$  then  $e_1$  else  $e_2$ ," our analysis doesn't consider whether  $e_0$  is true or false. This leads to situations where dead code (that is, if  $e_0$  is always true or always false) can cause the program to fail to type-check, or to a situation where the type system rejects a program that would have ill-defined behavior on a path that is logically impossible to take through the code. (We saw both of these situations in a previous quiz.)

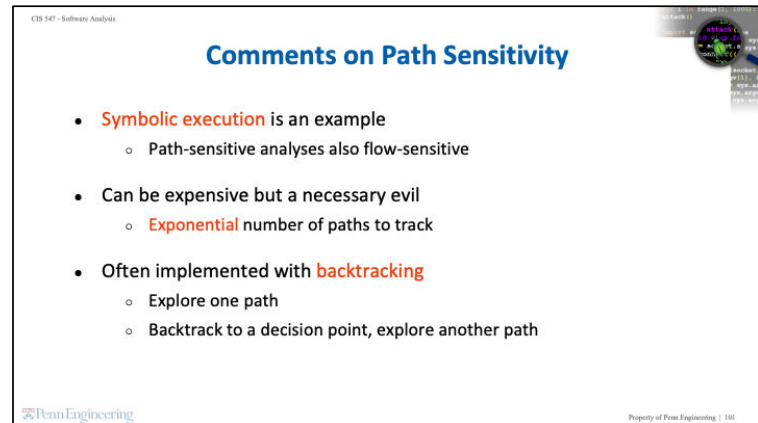
We call this sort of analysis "path-insensitive," as it does not consider whether certain paths through the code are impossible to take; it rejects a program that has a "bad" path, even if this path can never be taken (yielding false positives in the process). By making our analysis sensitive to program paths, we are able to eliminate this form of false positive at the cost of more complexity in our program abstraction.

To do so, we augment our environment for each statement with a boolean predicate  $P$ . If  $P$  is the predicate under which an [If-Then-Else] statement would be executed, then we add  $P$  to the input environment of the first hypothesis (that  $e_0$  is a bool). For the second hypothesis (that  $e_1$  has type  $t_1$ ), we add  $P$  conjoined with  $e_0$  to the input environment. And for the third hypothesis (that  $e_2$  has type  $t_2$ ), we add  $P$  conjoined with not- $e_0$  to the input environment. (Note that we are still retaining the distinction between input and output environments for all the hypotheses.)

Additionally, we no longer solve the side constraint that the output environments  $A_1$

and A2 are equal; instead, the conclusion allows for either of these to be its output environment based on the truth value of e0.

<#>



CIS 547 - Software Analysis

### Comments on Path Sensitivity

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

Penn Engineering Property of Penn Engineering | 101

Symbolic execution, which we will study later, is an example of a path-sensitive analysis. (Note that path-sensitive analyses are also flow-sensitive, as we preserve the input-output environment ordering.)

As you might imagine, keeping track of these different execution paths can be quite expensive. In the general case, there can be an exponential number of paths to track compared to the number of statements in the program! But this complexity appears to be a necessary evil in many different applications.

Path-sensitive analyses are often performed using backtracking instead of explicitly merging environments together. A single path is explored until termination, and then the analysis backtracks to a decision point before exploring another path. This allows clients to prioritize the exploration of paths that are potentially more interesting than others, for instance, in the case of a bug-finding tool, paths that are more likely to expose a bug.

CS 547 - Software Analysis

### QUIZ: Flow and Path Sensitivity

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

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

Penn Engineering Property of Penn Engineering | 102

{QUIZ SLIDE}

In this quiz, you will decide whether some sample programs can be proven to have certain properties by flow-insensitive, flow-sensitive, and path-sensitive analyses. For each of the following five programs, check the boxes corresponding to the kinds of analyses that can verify the specified property.

The first program consists of four statements executed in the following order: `x` is assigned the string literal "a"; `y` is assigned 5, `z` is assigned the result of `3 + y`, and `w` is assigned the result of `x +` the string literal "b". For this program, the property to verify is that there are no integer plus string errors. For this question, allow the analysis to assume that each variable is defined before being used, which is indeed the case for this program.

For the second and third programs, the property to verify is that there are no divide-by-zero errors. The second program consists of three statements executed in the following order: `x` is assigned 5; `y` is assigned 1 divided by `x`; and `x` is assigned 0. The third program consists of an if-then-else expression: if `y` does not equal zero, then 1 divided by `y`, else `y`. Note that the variable `y` may have any integer value at the start of the program.

For the next two programs, the property to verify is that locks are used correctly: that is, there is no attempt to acquire a lock on a resource already locked, and there is no

attempt to release a lock on a resource that is unlocked. The fourth program consists of two statements executed in the following order: `acquireLock(r)`; `releaseLock(r)`. Allow the analysis to assume that `r` is not locked at the start of the program. Finally, the fifth program consists of two if-then statements executed in the following order: if (`z > 0`) then `acquireLock(r)`; followed by if (`z > 0`) then `releaseLock(r)`. Note that the second if-then statement is not nested within the first if-then statement. Also note that variable `z` can have any integer value at the start of the program. Finally, as for the preceding program, allow the analysis to assume that `r` is not locked at the start of the program.

### QUIZ: Flow and Path Sensitivity

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

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

#### {SOLUTION SLIDE}

The first program can be proven to be free of integer plus string errors using a flow-insensitive analysis, assuming that all variables are defined in the program before being used.

Let's see why this is the case. A flow insensitive analysis views this program as an unordered set of these four statements, and effectively checks all possible orderings of these four statements for integer-plus-string errors. The analysis ignores orderings where a variable is used before being defined, since we allowed the analysis to make this assumption. Then, it is easy to see that on each remaining ordering, the analysis verifies that there is no integer-plus-string error. Since the original program represents one of these orderings, it follows that the flow-insensitive analysis has proven that it too is free of integer-plus-string errors.

Since flow-sensitive and path-sensitive analyses are strictly more powerful than flow-insensitive analysis, it follows that they too can prove this program free of integer-plus-string errors.

The second program cannot be proven by flow-insensitive analysis to be free of divide-by-zero errors. Because flow-insensitive analysis considers the statements as an unordered set, it does not know that the assignment of 0 to x comes after the assignment of 1/x to y. However, flow-sensitive analysis does consider statement

order, so it will know that x will not be 0 at the execution of the division operation. Because path-sensitive analysis is strictly more powerful than flow-sensitive analysis, it will also be able to verify that the program will not throw a division-by-zero error.



CS 547 - Software Analysis

### QUIZ: Flow and Path Sensitivity

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

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

Penn Engineering Property of Penn Engineering | 104

{SOLUTION SLIDE}

The third program cannot be proven even by flow-sensitive analysis to be free of divide-by-zero errors. Flow-sensitive analysis, while it does keep track of the order of statement execution, does not keep track of which branches are taken at any given decision point for the program. For example, if `1/y` and `y` had been switched so that the program read “if `y` does not equal 0, then `y`, else `1/y`”, a path-insensitive analysis could not tell the difference. All it would know is that either `y` or `1/y` could appear after the check that `y` doesn’t equal zero. Therefore, it could not prove that the division by `y` won’t throw an error. However, a path-sensitive analysis would be able to tell the difference if we were to swap the positions of `1/y` and `y` in the if-then-else expression. In particular, it could prove that if `1/y` is in the `y != 0` branch, then an error would not occur.

Now, for the fourth and fifth programs, the analysis must prove that the program does not acquire any locked resources or release any unlocked resources.

The fourth program relies on the particular statement execution order in order to ensure that the lock on `r` is not released while `r` is unlocked. Therefore, a flow-insensitive analysis could not prove that `r` is not improperly unlocked, while a flow-sensitive analysis (and therefore a path-sensitive analysis) could.

To verify that the fifth program does not have any locking errors, certainly requires at

least flow sensitivity to detect that any lock acquisition comes before lock release. However, just flow-sensitivity is not sufficient in this case, as it is too weak to recognize that if `z > 0`, the “then” portion of both branch points must be taken. Path-sensitive analysis would be able to recognize this fact, so path-sensitive analysis is able to prove that the program does not have any locking errors.

CS 547 - Software Analysis

## SEGMENT

### Summary

Penn Engineering Property of Penn Engineering | 105

CS 547 - Software Analysis

## Summary

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

Penn Engineering Property of Penn Engineering | 106

We have discussed different kinds of analyses and their abilities to detect errors in programs with different structures. While we have used the notation and terminology of type systems in this discussion, the typical applications of these analyses often have different names and purposes in practical applications.

While it is certainly possible to create a type system that is flow-sensitive or even path-sensitive, we normally think of type systems as flow-insensitive analyses. Flow-sensitive analyses are usually associated with dataflow analyses, and symbolic execution is a prototypical path-sensitive analysis.

However, these lines have become more blurred lately. For example, in the research literature, it is not uncommon to come across flow-sensitive type systems and path-sensitive dataflow analyses.



## LESSON

Review



## SEGMENT

What Have We Learned?

CIS 547 - Software Analysis

SEGMENT

What Have We Learned?

Penn Engineering

Property of Penn Engineering | 109

CIS 547 - Software Analysis

What Have We Learned?

- What is a **type**
- Computing types of programs using **type rules**
- Properties of type systems: **soundness, incompleteness, type inference**
- Describing other analyses using types notation
- Classifying analyses: **flow-insensitive vs. flow-sensitive vs. path-sensitive**

Penn Engineering

Property of Penn Engineering | 110

This concludes our introduction to type systems and their usefulness to software analysis in general.

In this module, we have learned:

What a “type” is: an abstract value consisting of a set of concrete values having that type;  
How to compute the types of programs using derivations from type rules;  
The properties of type systems: soundness (that is, guaranteeing no false negatives), incompleteness (that is, the occurrence of false positives), and type inference;  
How to describe other kinds of static analyses using the notation of type systems; and  
The strengths and costs of flow-insensitive, flow-sensitive, and path-sensitive analyses.

We have only scratched the surface of type systems in this module. The study of type systems is an active area of research with many fruitful results. We have included a list of resources below for further reading on topics in type systems.

[Article titled “Type Systems” by Luca Cardelli in CRC Handbook:  
<http://www.cc.gatech.edu/~naik/courses/cs6340/papers/cardelli-types.pdf>]

[Book titled “Types and Programming Languages” by Benjamin Pierce:  
<https://www.cis.upenn.edu/~bcpierce/tapl/>]

[Book titled “Advanced Topics in Types and Programming Languages”  
edited by Benjamin Pierce: <https://www.cis.upenn.edu/~bcpierce/attapl/>]