

L29: Type Systems

Software Analysis

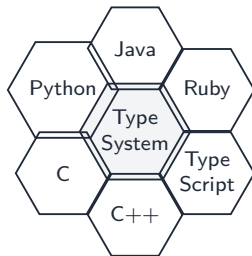
IIT Guwahati · October 14, 2026

Type Systems (1/2)

- Most widely used form of static analysis

Type Systems (2/2)

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



Naik, CIS 5470, Penn: Type Systems

Motivation (1/2)

T.java

```
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: }
```

Motivation (2/2)

T.java

```
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: }
```

```
$ 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
```

Line 4: the condition `a` is a `float`, not a `boolean`. Line 7: `c` is an `int[]`, but `f` must return an `int`.

Motivation in C++ (1/3)

T.cpp

```
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: };
```

The same function in C++, checked by the clang compiler.

Motivation in C++ (2/3)

T.cpp

```
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: };
```

Only **one** error, on line 7: the array `c` is a pointer `int *`, which cannot be returned as an `int`.

Line 4 is legal C++: the `float a` is converted to `bool`.

```
$ clang++ -fsyntax-only T.cpp  
T.cpp:7:14: error: cannot initialize return object of type 'int' with an lvalue of type 'int *'  
    7 |         return c;  
      |         ^  
1 error generated.
```

Motivation in C++ (3/3)

T.cpp

```
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: };
```

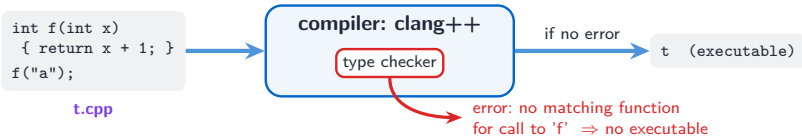
With `-Wfloat-conversion`, clang reports line 4 as a **warning**, not an error.

Each language has its own type rules.

```
$ clang++ -fsyntax-only -Wfloat-conversion T.cpp
T.cpp:4:9: warning: implicit conversion turns floating-point number into integer: 'float' to 'bool' [-Wfloat-conversion]
    4 |         if (a)
      |         ~~ ^
T.cpp:7:14: error: cannot initialize return object of type 'int' with an lvalue of type 'int *'
    7 |         return c;
      |         ^
1 warning and 1 error generated.
```

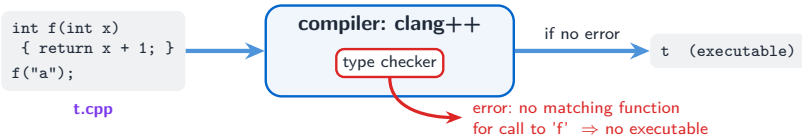
Where the Type System Runs (1/3)

C++ (statically typed)



Where the Type System Runs (2/3)

C++ (statically typed)

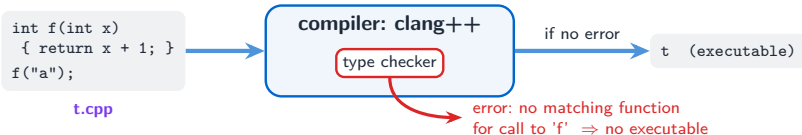


Python (dynamically typed)



Where the Type System Runs (3/3)

C++ (statically typed)



Python (dynamically typed)



Static typing: types checked *before* the program runs, for every path.

Dynamic typing: types checked *while* it runs, only on the operations that execute.

Static and Dynamic Checking: Example (1/4)

t.cpp

```
1 #include <stdio>
2 int f(int x) { return x + 1; }
3
4 int main() {
5     std::puts("start");
6     std::printf("%d\n", f(2));
7     std::printf("%d\n", f("a"));
8 }
```

t.py

```
1 def f(x):
2     return x + 1
3
4 print("start")
5 print(f(2))
6 print(f("a"))
```

Naik, CIS 5470, Penn: Type Systems

Static and Dynamic Checking: Example (2/4)

t.cpp

```
1 #include <stdio>
2 int f(int x) { return x + 1; }
3
4 int main() {
5     std::puts("start");
6     std::printf("%d\n", f(2));
7     std::printf("%d\n", f("a"));
8 }
```

t.py

```
1 def f(x):
2     return x + 1
3
4 print("start")
5 print(f(2))
6 print(f("a"))
```

```
$ clang++ t.cpp -o t
t.cpp:7:23: error: no matching function for call to 'f'
    7 |     std::printf("%d\n", f("a"));
      |                        ^
t.cpp:2:5: note: candidate function not viable: no known conversion from 'const char[2]' to 'int' for
1st argument
    2 | int f(int x) { return x + 1; }
      |     ^~~~~~
1 error generated.
```

C++: rejected at compile time. "start" is never printed.

Static and Dynamic Checking: Example (3/4)

t.cpp

```
1 #include <stdio>
2 int f(int x) { return x + 1; }
3
4 int main() {
5     std::puts("start");
6     std::printf("%d\n", f(2));
7     std::printf("%d\n", f("a"));
8 }
```

t.py

```
1 def f(x):
2     return x + 1
3
4 print("start")
5 print(f(2))
6 print(f("a"))
```

```
$ python3 - < t.py
start
3
Traceback (most recent call last):
  File "<stdin>", line 6, in <module>
    File "<stdin>", line 2, in f
TypeError: can only concatenate str (not "int") to str
```

Python: start and 3 are printed, then the error, when line 6 runs.

Static and Dynamic Checking: Example (4/4)

t.cpp

```
1 #include <stdio>
2 int f(int x) { return x + 1; }
3
4 int main() {
5     std::puts("start");
6     std::printf("%d\n", f(2));
7     std::printf("%d\n", f("a"));
8 }
```

t.py

```
1 def f(x):
2     return x + 1
3
4 print("start")
5 print(f(2))
6 print(f("a"))
```

```
$ python3 - < t.py
start
3
Traceback (most recent call last):
  File "<stdin>", line 6, in <module>
    File "<stdin>", line 2, in f
TypeError: can only concatenate str (not "int") to str
```

Same bug: C++ finds it **before** running; Python only **when** that line runs.

Type Systems as a Notation (1/2)

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

Type Systems as a Notation (2/2)

- Most widely used form of static analysis
- Part of nearly all mainstream languages
- **Provides notation useful for describing static analyses**
type checking, dataflow analysis, symbolic execution, ...

What Is a Type? (1/2)

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

What Is a Type? (2/2)

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

- Examples in Java:

`int`: the set of all integers between -2^{31} and $(2^{31})-1$

`double`: the set of all double-precision floating point numbers

`boolean`: the set `{true, false}`

More Examples (1/3)

- Foo is the set of all objects of class Foo

More Examples (2/3)

- 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**: a function from types to types

More Examples (3/3)

- `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**: a function from types to types
- `int -> int` is the set of functions taking an `int` and returning an `int`
E.g.: `increment`, a function that squares a number, etc.

Abstraction (1/3)

- All static analyses use **abstraction**

Represent sets of concrete values as abstract values

Abstraction (2/3)

- All static analyses use **abstraction**

Represent sets of concrete values as abstract values

- **Why?**

Cannot directly reason about infinite sets of concrete values (no guarantee of **termination**)¹

Improves **performance** even for (large) finite sets

¹ Static analysis is *undecidable*: no algorithm can answer an interesting question about program behaviour exactly, for every program, and always terminate (Rice's theorem). So every analysis must approximate.

Abstraction (3/3)

- All static analyses use **abstraction**

Represent sets of concrete values as abstract values

- **Why?**

Cannot directly reason about infinite sets of concrete values (no guarantee of **termination**)¹

Improves **performance** even for (large) finite sets

- In type systems, the abstractions are called **types**

¹ Static analysis is *undecidable*: no algorithm can answer an interesting question about program behaviour exactly, for every program, and always terminate (Rice's theorem). So every analysis must approximate.

Types as Abstract Values (1/2)

- A type is an example of an **abstract value**
Represents a **set** of **concrete** values

Types as Abstract Values (2/2)

- 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

The Next Steps

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

The Lambda Calculus (1/9)

(expression)	$e := v \mid x \mid e1 + e2 \mid e1\ e2$	
(value)	$v := i \mid \lambda x:t \Rightarrow e$	
(integer)	i	
(variable)	x	
(type)	$t := \text{int} \mid t1 \rightarrow t2$	

i : the integer itself,
already a value



Lambda calculus: a minimal language in which everything is a function, introduced by Alonzo Church in the 1930s. $\lambda x:t \Rightarrow e$ is a function without a name: its parameter is x of type t , and its body is e .

The Lambda Calculus (2/9)

(expression) $e := v \mid x \mid e1 + e2 \mid e1\ e2$

(value) $v := i \mid \lambda x:t \Rightarrow e$

(integer) i

(variable) x

(type) $t := \text{int} \mid t1 \rightarrow t2$

$\lambda x:t \Rightarrow e$: already a value, a function; its body e is not run yet



Lambda calculus: a minimal language in which everything is a function, introduced by Alonzo Church in the 1930s. $\lambda x:t \Rightarrow e$ is a function without a name: its parameter is x of type t , and its body is e .

The Lambda Calculus (3/9)

(expression)	$e := v \mid x \mid e1 + e2 \mid e1\ e2$	x : the value bound to x when a function was called
(value)	$v := i \mid \lambda x:t \Rightarrow e$	
(integer)	i	
(variable)	x	
(type)	$t := \text{int} \mid t1 \rightarrow t2$	

Lambda calculus: a minimal language in which everything is a function, introduced by Alonzo Church in the 1930s. $\lambda x:t \Rightarrow e$ is a function without a name: its parameter is x of type t , and its body is e .

The Lambda Calculus (4/9)

(expression)	$e := v \mid x \mid e1 + e2 \mid e1\ e2$	$e1 + e2$: evaluate both to integers n_1, n_2 ; the result is $n_1 + n_2$
(value)	$v := i \mid \lambda x:t \Rightarrow e$	
(integer)	i	
(variable)	x	
(type)	$t := \text{int} \mid t1 \rightarrow t2$	

Lambda calculus: a minimal language in which everything is a function, introduced by Alonzo Church in the 1930s. $\lambda x:t \Rightarrow e$ is a function without a name: its parameter is x of type t , and its body is e .

The Lambda Calculus (5/9)

(expression)	$e := v \mid x \mid e1 + e2 \mid e1\ e2$	 $e1\ e2$: evaluate $e1$ to $\lambda x:t \Rightarrow e$ and $e2$ to a value v ; then evaluate e with x replaced by v
(value)	$v := i \mid \lambda x:t \Rightarrow e$	
(integer)	i	
(variable)	x	
(type)	$t := \text{int} \mid t1 \rightarrow t2$	

Lambda calculus: a minimal language in which everything is a function, introduced by Alonzo Church in the 1930s. $\lambda x:t \Rightarrow e$ is a function without a name: its parameter is x of type t , and its body is e .

The Lambda Calculus (6/9)

(expression) $e := v \mid x \mid e1 + e2 \mid e1\ e2$

(value) $v := i \mid \lambda x:t \Rightarrow e$

(integer) i

(variable) x

(type) $t := \text{int} \mid t1 \rightarrow t2$

int : all integers.

$t1 \rightarrow t2$: all functions
from $t1$ values to $t2$
values

Lambda calculus: a minimal language in which everything is a function, introduced by Alonzo Church in the 1930s. $\lambda x:t \Rightarrow e$ is a function without a name: its parameter is x of type t , and its body is e .

The Lambda Calculus (7/9)

(expression) $e := v \mid x \mid e1 + e2 \mid e1\ e2$
(value) $v := i \mid \lambda x:t \Rightarrow e$
(integer) i
(variable) x
(type) $t := \text{int} \mid t1 \rightarrow t2$

Example Program: $(\lambda x:\text{int} \Rightarrow (x + 1))\ (42)$

the same as `int f(int x) { return x + 1; } then f(42)`

Lambda calculus: a minimal language in which everything is a function, introduced by Alonzo Church in the 1930s. $\lambda x:t \Rightarrow e$ is a function without a name: its parameter is x of type t , and its body is e .

The Lambda Calculus (8/9)

(expression) $e := v \mid x \mid e1 + e2 \mid e1\ e2$

(value) $v := i \mid \lambda x:t \Rightarrow e$

(integer) i

(variable) x

(type) $t := \text{int} \mid t1 \rightarrow t2$

$(\lambda x:\text{int} \Rightarrow (x + 1))\ (42) \rightarrow 42 + 1 \rightarrow 43$

Lambda calculus: a minimal language in which everything is a function, introduced by Alonzo Church in the 1930s. $\lambda x:t \Rightarrow e$ is a function without a name: its parameter is x of type t , and its body is e .

The Lambda Calculus (9/9)

(expression) $e := v \mid x \mid e1 + e2 \mid e1\ e2$

(value) $v := i \mid \lambda x:t \Rightarrow e$

(integer) i

(variable) x

(type) $t := \text{int} \mid t1 \rightarrow t2$

$(\lambda x:\text{int} \Rightarrow (x + 1))\ (42) \rightarrow 42 + 1 \rightarrow 43$

$42\ (\lambda x:\text{int} \Rightarrow (x + 5)) \rightarrow$ **stuck**: 42 is not a function

Lambda calculus: a minimal language in which everything is a function, introduced by Alonzo Church in the 1930s. $\lambda x:t \Rightarrow e$ is a function without a name: its parameter is x of type t , and its body is e .

QUIZ: Programs and Types (1/2)

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

Program	Type
<code>λ x:int => (x + x)</code>	
<code>(λ x:int => (x + x)) (10)</code>	
<code>42 (λ x:int => (x + 5))</code>	
<code>λ x:int => (λ y:int => (x + y))</code>	
<code>(λ x:int => x) + 10</code>	

QUIZ: Programs and Types (2/2)

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

Program	Type
<code>λ x:int => (x + x)</code>	<code>int -> int</code>
<code>(λ x:int => (x + x)) (10)</code>	<code>int</code>
<code>42 (λ x:int => (x + 5))</code>	NONE: 42 is not a function
<code>λ x:int => (λ y:int => (x + y))</code>	<code>int -> (int -> int)</code>
<code>(λ x:int => x) + 10</code>	NONE: + on a function

Notation for Inference Rules (1/3)

- Inference rules have the following form:

If (hypothesis) is true, then (conclusion) is true

Notation for Inference Rules (2/3)

- Inference rules have the following form:

If (hypothesis) is true, then (conclusion) is true

- Type checking computes via reasoning:

If e_1 is an int and e_2 is a double, then $e_1 * e_2$ is a double

Notation for Inference Rules (3/3)

- Inference rules have the following form:

If (hypothesis) is true, then (conclusion) is true

- Type checking computes via reasoning:

If e_1 is an int and e_2 is a double, then $e_1 * e_2$ is a double

- We will develop a standard notation for inference rules

From English to Inference Rule (1/5)

- 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 ”

From English to Inference Rule (2/5)

- If $e1$ has type `int` and $e2$ has type `int`, then $e1 + e2$ has type `int`

From English to Inference Rule (3/5)

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

From English to Inference Rule (4/5)

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

From English to Inference Rule (5/5)

The statement

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

is a special case of

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

Writing Inference Rules (1/3)

- By tradition, inference rules are written

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

Writing Inference Rules (2/3)

- By tradition, inference rules are written

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

- Hypotheses and conclusion are **type judgments**:

$$\vdash e : t$$

Writing Inference Rules (3/3)

- By tradition, inference rules are written

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

- Hypotheses and conclusion are **type judgments**:

$$\vdash e : t$$

- $\vdash$ means “it is provable that...”
- A rule with no hypotheses is an **axiom**

Rules for Integers (1/6)

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

Axiom: no hypotheses
above the line, so its
conclusion always holds

Rules for Integers (2/6)

$$\frac{}{\vdash i : \text{int}} \text{ [Int]}$$
$$\frac{\vdash e1 : \text{int} \quad \vdash e2 : \text{int}}{\vdash e1 + e2 : \text{int}}$$

[Add] ← **Inference rule:** if the hypotheses hold, the conclusion holds

Rules for Integers (3/6)

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

Hypotheses: above the line, joined by “and”

Conclusion: below the line

Rules for Integers (4/6)

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

- Templates (**schemas**) for how to type integers and sums

Rules for Integers (5/6)

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

- Templates (**schemas**) for how to type integers and sums
- Filling in templates produces complete typings

Rules for Integers (6/6)

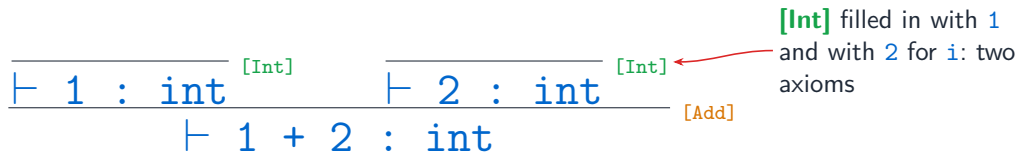
$$\frac{}{\vdash i : \text{int}} \text{[Int]}$$
$$\frac{\vdash e1 : \text{int} \quad \vdash e2 : \text{int}}{\vdash e1 + e2 : \text{int}} \text{[Add]}$$

- Templates (**schemas**) 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**

Example: $1 + 2$ (1/2)



Example: $1 + 2$ (2/2)

$$\frac{\overline{\vdash 1 : \text{int}}^{\text{[Int]}} \quad \overline{\vdash 2 : \text{int}}^{\text{[Int]}}}{\vdash 1 + 2 : \text{int}}^{\text{[Add]}}$$

[Add]: the two conclusions above become its hypotheses

Naik, CIS 5470, Penn: Type Systems

A Problem (1/2)

What is the type of a variable reference?

$$\frac{\vdash e : \text{int}}{\vdash e + e : \text{int}}$$

Carries type information
for e in hypotheses

A Problem (2/2)

What is the type of a variable reference?

$$\frac{\vdash e : \text{int}}{\vdash e + e : \text{int}}$$

Carries type information
for e in hypotheses

$$\frac{}{\vdash x : ?} \text{ [Var]}$$

Doesn't carry enough
information to give x a type

A Solution (1/2)

- Put more information in the rules!

$$\frac{}{\vdash x : ?} \text{[Var]}$$

A Solution (2/2)

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

A variable is **free** in an expression if not defined within the expression; otherwise it is **bound**

e.g. in $\lambda x:\text{int} \Rightarrow (x + y)$: x is bound, y is free

An **environment** is a function from variables to types

May map variables to other abstract values in different static analyses

$$\frac{}{\vdash x : ?} \text{ [Var]}$$

Type Environments (1/3)

- Let A be a function from `variables` to `types`

Type Environments (2/3)

- Let A be a function from `variables` to `types`
- The sentence $A \vdash e : t$ means:
“Under the assumption that variables have types given by A , it is provable that expression e has type t .”

Type Environments (3/3)

- Let A be a function from `variables` to `types`
- The sentence $A \vdash e : t$ means:
“Under the assumption that variables have types given by A , it is provable that expression e has type t .”
- Example: $[x \mapsto \text{int}] \vdash x + 1 : \text{int}$
if x has type `int`, then $x + 1$ has type `int`

Modified Rules

- The type environment is added to all rules:

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

A New Rule

- And we can write new rules:

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

$A(x)$: the type that A gives to x ; e.g. with $A = [x \mapsto \text{int}]$, $x : \text{int}$

Rules for Functions (1/5)

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

whole function:
x is **bound**

$\lambda x:\text{int} \Rightarrow (x + 1)$

the body e alone: x is **free**,
so A has no type for x

Rules for Functions (2/5)

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

$A[x \mapsto t]$ means “ A modified to map x to type t ”: type the body with the parameter added

Rules for Functions (3/5)

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

$A[x \mapsto t]$ means “ A modified to map x to type t ”: type the body with the parameter added

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

$e1$ is called with argument $e2$

Rules for Functions (4/5)

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

$A[x \mapsto t]$ means “ A modified to map x to type t ”: type the body with the parameter added

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

the argument $e2$ must have the parameter type $t1$ of $e1$

Rules for Functions (5/5)

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

$A[x \mapsto t]$ means “ A modified to map x to type t ”: type the body with the parameter added

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

the call has the result type $t2$ of $e1$

All Rules Together

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

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

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

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

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

Type Derivations: Example (1/8)

environment:
empty, no variables

expression: the program

$[] \vdash (\lambda x:\text{int} \Rightarrow (x + 1)) (42) : \text{int}$

“it is provable that”

its type

Goal: prove this judgment. We work from the bottom up.

Type Derivations: Example (2/8)

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

[Call]: e_1 is the function, e_2 is 42, and $t_1 = t_2 = \text{int}$.

So we must prove the function has type $\text{int} \rightarrow \text{int}$ and 42 has type int .

Type Derivations: Example (3/8)

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

[Int] is the axiom $\frac{}{A \vdash i : \text{int}} [\text{Int}]$: no hypotheses above the line.

Fill it in with $A := []$ and $i := 42$: it gives $[] \vdash 42 : \text{int}$. Nothing more to prove.

Type Derivations: Example (4/8)

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

[Def]: the function $\lambda x:\text{int} \Rightarrow (x + 1)$. Type its body with $x \mapsto \text{int}$ added to the environment: prove $x + 1 : \text{int}$.

Type Derivations: Example (5/8)

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

[Add]: $x + 1$ is a sum. Prove both operands are `int`, in the same environment.

Type Derivations: Example (6/8)

$$\begin{array}{c}
 \frac{[x \mapsto \text{int}] \vdash x : \text{int} \quad \frac{[x \mapsto \text{int}] \vdash 1 : \text{int} \quad [\text{Int}]}{[x \mapsto \text{int}] \vdash x + 1 : \text{int}} [\text{Add}]}{[x \mapsto \text{int}] \vdash x + 1 : \text{int}} [\text{Def}] \quad \frac{[\text{Int}]}{[\text{Int}]} [\text{Int}] \\
 \hline
 \frac{[\text{Int}] \vdash \lambda x:\text{int} \Rightarrow (x + 1) : \text{int} \rightarrow \text{int} \quad [\text{Int}] \vdash 42 : \text{int} \quad [\text{Int}]}{[\text{Int}] \vdash (\lambda x:\text{int} \Rightarrow (x + 1)) (42) : \text{int}} [\text{Call}]
 \end{array}$$

[Int]: 1 is an integer literal, whatever the environment. An axiom.

Type Derivations: Example (7/8)

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

[Var]: the environment maps x to int , so $x : \text{int}$. An axiom.

Type Derivations: Example (8/8)

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

Every branch ends in an axiom: the program has type `int`.

QUIZ: Type Derivations (1/6)

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

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

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

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

QUIZ: Type Derivations (2/6)

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

:

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

:

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

:

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

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

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

Step 1: **[Def]**: the body $\lambda y:\text{int} \Rightarrow (x + y)$ must have type $\text{int} \rightarrow \text{int}$

QUIZ: Type Derivations (3/6)

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

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

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

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

Step 2: [Def] again: add $y \mapsto \text{int}$; the body $x + y$ must have type int

QUIZ: Type Derivations (4/6)

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

x

:

int

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

:

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

$x + y$

:

int

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

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

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

Step 3: **[Add]**: the left operand x must be int

QUIZ: Type Derivations (5/6)

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

x

:

int

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

y

:

int

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

$x + y$

:

int

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

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

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

Step 3: **[Add]**: the right operand y must be int

QUIZ: Type Derivations (6/6)

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

x

:

int

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

y

:

int

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

$x + y$

:

int

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

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

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

Rules used, bottom up: [Def], [Def], [Add], then [Var] twice.