L32: Type Systems (Continued)

Software Analysis

IIT Guwahati · Oct 21

Type Constraints

$$\frac{A \vdash e0 : bool \quad A \vdash e1 : t1 \quad A \vdash e2 : t2 \quad t1 = t2}{A \vdash \texttt{if } e0 \texttt{ then } e1 \texttt{ else } e2 : t1}$$

Side constraints must be solved

Many analyses have **side conditions**

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

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

then: `int -> int` else: `int`

so `t1 = t2` fails: not typeable

Another Example (1/3)

- Consider a **recursive function**

  ```
  f(x) = ... f(e) ...
  ```

  e.g. `fact(n) = if n == 0 then 1 else n * fact(n - 1)`

Another Example (2/3)

- Consider a **recursive function**

  `f(x) = ... f(e) ...`

  e.g. `fact(n) = if n == 0 then 1 else n * fact(n - 1)`

- If `x : t1` and `e : t2` then `t2 = t1`

  `e` is passed for `x`; in `fact`: `[[n - 1]] = [[n]]`

  Can be relaxed to $\texttt{t2} \subseteq \texttt{t1}$ (a type is a set of values: *subtyping*)

Another Example (3/3)

- Consider a **recursive function**

  `f(x) = ... f(e) ...`

  e.g. `fact(n) = if n == 0 then 1 else n * fact(n - 1)`

- If `x : t1` and `e : t2` then `t2 = t1`

  `e` is passed for `x`; in `fact`: `[[n - 1]] = [[n]]`

  Can be relaxed to `t2` $\subseteq$ `t1` (a type is a set of values: *subtyping*)

- Recursive functions yield **recursive constraints**

  Same with **loops**

  Hardness depends on the constraint language and application

Type Inference (1/4)

- Languages do not require types for entire programs to reduce verbosity

Type Inference (2/4)

- Languages do not require types for entire programs to reduce verbosity
- **Type inference** determines the type of a program, if it has one

Type Inference (3/4)

- 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 reconstruction**: it fills in the types the programmer left out

Naik, CIS 5470, Penn: Type Systems; Milner, JCSS 1978

Type Inference (4/4)

- 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 reconstruction**: it fills in the types the programmer left out
- A challenging problem for complex type systems and tricky languages

  $\Rightarrow$ many tradeoffs (sound vs. complete vs. efficient vs. automatic ...)

Naik, CIS 5470, Penn: Type Systems; Milner, JCSS 1978

Outline of Type Inference Algorithm (1/3)

- Introduce a type variable (denoted `[[..]]`) for each program fragment

  Variable, e.g. `[[x]]`, and expression, e.g. `[[e1]]`

Outline of Type Inference Algorithm (2/3)

- Introduce a type variable (denoted `[[..]]`) for each program fragment

  Variable, e.g. `[[x]]`, and expression, e.g. `[[e1]]`

- **Step 1**: Generate constraints based on the type rules

Naik, CIS 5470, Penn: Type Systems; Robinson, JACM 1965

Outline of Type Inference Algorithm (3/3)

- Introduce a type variable (denoted `[[..]]`) for each program fragment

  Variable, e.g. `[[x]]`, and expression, e.g. `[[e1]]`

- **Step 1**: Generate constraints based on the type rules
- **Step 2**: Solve the constraints to obtain types or show program is untypable

  Unification algorithm $\rightarrow$ automated/interactive theorem prover

Naik, CIS 5470, Penn: Type Systems; Robinson, JACM 1965

Step 1: Constraints from the Rules (1/5)

Each rule used in a derivation gives these constraints:

Rule	Fragment	Constraints generated
`[Int]`	integer `i`	`[[i]] = int`

Step 1: Constraints from the Rules (2/5)

Each rule used in a derivation gives these constraints:

Rule	Fragment	Constraints generated
`[Int]`	integer `i`	`[[i]] = int`
`[Var]`	variable `x`	none: every use of `x` shares `[[x]]`

Step 1: Constraints from the Rules (3/5)

Each rule used in a derivation gives these constraints:

Rule	Fragment	Constraints generated
`[Int]`	integer `i`	`[[i]] = int`
`[Var]`	variable `x`	none: every use of `x` shares `[[x]]`
`[Add]`	`e1 + e2`	`[[e1]] = int`, `[[e2]] = int`, `[[e1 + e2]] = int`

Step 1: Constraints from the Rules (4/5)

Each rule used in a derivation gives these constraints:

Rule	Fragment	Constraints generated
[Int]	integer `i`	`[[i]] = int`
[Var]	variable `x`	none: every use of `x` shares `[[x]]`
[Add]	`e1 + e2`	`[[e1]] = int`, `[[e2]] = int`, `[[e1 + e2]] = int`
[Def]	`λ x:t => e`	`[[x]] = t`, `[[λ x:t => e]] = [[x]] -> [[e]]`

Step 1: Constraints from the Rules (5/5)

Each rule used in a derivation gives these constraints:

Rule	Fragment	Constraints generated
`[Int]`	integer `i`	`[[i]] = int`
`[Var]`	variable `x`	none: every use of `x` shares `[[x]]`
`[Add]`	`e1 + e2`	`[[e1]] = int`, `[[e2]] = int`, `[[e1 + e2]] = int`
`[Def]`	`λ x:t => e`	`[[x]] = t`, `[[λ x:t => e]] = [[x]] -> [[e]]`
`[Call]`	`e1 e2`	`[[e1]] = [[e2]] -> [[e1 e2]]`

Step 2: Unification (1/9)

Repeat, until no constraint is left:

1. `T = T`: drop it (trivially true)

Robinson, JACM 1965; Martelli and Montanari, TOPLAS 1982

Step 2: Unification (2/9)

Repeat, until no constraint is left:

1. `T = T`: drop it (trivially true)
2. `[[v]] = T`: **substitute** `T` for `[[v]]` in all other constraints (fail if `[[v]]` occurs inside `T`)

Robinson, JACM 1965; Martelli and Montanari, TOPLAS 1982

Step 2: Unification (3/9)

Repeat, until no constraint is left:

1. `T = T`: drop it (trivially true)
2. `[[v]] = T`: **substitute** `T` for `[[v]]` in all other constraints (fail if `[[v]]` occurs inside `T`)
3. `T1 -> T2 = T3 -> T4`: **split** into `T1 = T3` and `T2 = T4`

Robinson, JACM 1965; Martelli and Montanari, TOPLAS 1982

Step 2: Unification (4/9)

Repeat, until no constraint is left:

1. `T = T`: drop it (trivially true)
2. `[[v]] = T`: **substitute** `T` for `[[v]]` in all other constraints (fail if `[[v]]` occurs inside `T`)
3. `T1 -> T2 = T3 -> T4`: **split** into `T1 = T3` and `T2 = T4`
4. `int = T1 -> T2`: **fail** (different type constructors)

Robinson, JACM 1965; Martelli and Montanari, TOPLAS 1982

Step 2: Unification (5/9)

Repeat, until no constraint is left:

1. `T = T`: drop it (trivially true)
2. `[[v]] = T`: **substitute** `T` for `[[v]]` in all other constraints (fail if `[[v]]` occurs inside `T`)
3. `T1 -> T2 = T3 -> T4`: **split** into `T1 = T3` and `T2 = T4`
4. `int = T1 -> T2`: **fail** (different type constructors)

Example:

```
[[x]] -> int = int -> [[y]]
```

Robinson, JACM 1965; Martelli and Montanari, TOPLAS 1982

Step 2: Unification (6/9)

Repeat, until no constraint is left:

1. `T = T`: drop it (trivially true)
2. `[[v]] = T`: **substitute** `T` for `[[v]]` in all other constraints (fail if `[[v]]` occurs inside `T`)
3. `T1 -> T2 = T3 -> T4`: **split** into `T1 = T3` and `T2 = T4`
4. `int = T1 -> T2`: **fail** (different type constructors)

Example:

`[[x]] -> int = int -> [[y]]`

split: `[[x]] = int`, `int = [[y]]`

Robinson, JACM 1965; Martelli and Montanari, TOPLAS 1982

Step 2: Unification (7/9)

Repeat, until no constraint is left:

1. `T = T`: drop it (trivially true)
2. `[[v]] = T`: **substitute** `T` for `[[v]]` in all other constraints (fail if `[[v]]` occurs inside `T`)
3. `T1 -> T2 = T3 -> T4`: **split** into `T1 = T3` and `T2 = T4`
4. `int = T1 -> T2`: **fail** (different type constructors)

Example:

`[[x]] -> int = int -> [[y]]`

split: `[[x]] = int`, `int = [[y]]`

substitute: `[[x]] := int`, `[[y]] := int`

Robinson, JACM 1965; Martelli and Montanari, TOPLAS 1982

Step 2: Unification (8/9)

Repeat, until no constraint is left:

1. `T = T`: drop it (trivially true)
2. `[[v]] = T`: **substitute** `T` for `[[v]]` in all other constraints (fail if `[[v]]` occurs inside `T`)
3. `T1 -> T2 = T3 -> T4`: **split** into `T1 = T3` and `T2 = T4`
4. `int = T1 -> T2`: **fail** (different type constructors)

Example:

`[[x]] -> int = int -> [[y]]`

split: `[[x]] = int`, `int = [[y]]`

substitute: `[[x]] := int`, `[[y]] := int`

solved: both sides are `int -> int`

Robinson, JACM 1965; Martelli and Montanari, TOPLAS 1982

Step 2: Unification (9/9)

Repeat, until no constraint is left:

1. `T = T`: drop it (trivially true)
2. `[[v]] = T`: **substitute** `T` for `[[v]]` in all other constraints (fail if `[[v]]` occurs inside `T`)
3. `T1 -> T2 = T3 -> T4`: **split** into `T1 = T3` and `T2 = T4`
4. `int = T1 -> T2`: **fail** (different type constructors)

Example:

`[[x]] -> int = int -> [[y]]`

split: `[[x]] = int`, `int = [[y]]`

substitute: `[[x]] := int`, `[[y]] := int`

solved: both sides are `int -> int`

`int = [[y]] -> int`: **fail** (rule 4)

Robinson, JACM 1965; Martelli and Montanari, TOPLAS 1982

Type Inference: Example 1 (1/13)

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

Type Inference: Example 1 (2/13)

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

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

1.
2.
3.

```
[[x]] = int
[[y]] = int
[[(x + y)]] = int
```

`[Add]` for `x + y` gives constraints 1–3

Type Inference: Example 1 (3/13)

λ x:int => (λ y:int => (x + y))

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

1. [[x]] = int
2. [[y]] = int
3. [[(x + y)]] = int
4. [[(λ y:int => (x + y))]] = [[y]] -> [[(x + y)]]

[Def] for the inner function gives constraint 4

Type Inference: Example 1 (4/13)

λ x:int => (λ y:int => (x + y))

$$\frac{\texttt{A[x}\mapsto\texttt{t]} \vdash \texttt{e : t'}}{\texttt{A} \vdash \lambda\ \texttt{x:t => e : t -> t'}}\ \texttt{[Def]}$$

1. [[x]] = int
2. [[y]] = int
3. [[(x + y)]] = int
4. [[(λ y:int => (x + y))]] = [[y]] -> [[(x + y)]]
5. [[λ x:int => (λ y:int => (x + y))]] = [[x]] -> [[(λ y:int => (x + y))]]

[Def] for the whole program gives constraint 5. Now let's solve the constraints ...

Type Inference: Example 1 (5/13)

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

1. `[[x]] = int`
2. `[[y]] = int`
3. `[[(x + y)]] = int`
4. `[[(λ y:int => (x + y))]] = [[y]] -> [[(x + y)]]`
5. `[[λ x:int => (λ y:int => (x + y))]] = [[x]] -> [[(λ y:int => (x + y))]]`

Use constraint 1: replace `[[x]]` by `int` in constraint 5

Type Inference: Example 1 (6/13)

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

1. `[[x]] = int`
2. `[[y]] = int`
3. `[[(x + y)]] = int`
4. `[[(λ y:int => (x + y))]] = [[y]] -> [[(x + y)]]`
5. `[[λ x:int => (λ y:int => (x + y))]] = int -> [[(λ y:int => (x + y))]]`

Use constraint 1: replace `[[x]]` by `int` in constraint 5

Type Inference: Example 1 (7/13)

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

```
1.                                                    [[x]] = int
2.                                                    [[y]] = int
3.                                                [[(x + y)]] = int
4.                        [[(λ y:int => (x + y))]] = [[y]] -> [[(x + y)]]
5. [[λ x:int => (λ y:int => (x + y))]] = int -> [[(λ y:int => (x + y))]]
```

Use constraint 2: replace `[[y]]` by `int` in constraint 4

Type Inference: Example 1 (8/13)

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

```
1.                                                    [[x]] = int
2.                                                    [[y]] = int
3.                                                [[(x + y)]] = int
4.                          [[(λ y:int => (x + y))]] = int -> [[(x + y)]]
5. [[λ x:int => (λ y:int => (x + y))]] = int -> [[(λ y:int => (x + y))]]
```

Use constraint 2: replace `[[y]]` by `int` in constraint 4

Type Inference: Example 1 (9/13)

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

1. `[[x]] = int`
2. `[[y]] = int`
3. `[[(x + y)]] = int`
4. `[[(λ y:int => (x + y))]] = int -> [[(x + y)]]`
5. `[[λ x:int => (λ y:int => (x + y))]] = int -> [[(λ y:int => (x + y))]]`

Use constraint 3: replace `[[(x + y)]]` by `int` in constraint 4

Type Inference: Example 1 (10/13)

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

1. `[[x]] = int`
2. `[[y]] = int`
3. `[[(x + y)]] = int`
4. `[[(λ y:int => (x + y))]] = int -> int`
5. `[[λ x:int => (λ y:int => (x + y))]] = int -> [[(λ y:int => (x + y))]]`

Use constraint 3: replace `[[(x + y)]]` by `int` in constraint 4

Type Inference: Example 1 (11/13)

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

1. `[[x]] = int`
2. `[[y]] = int`
3. `[[(x + y)]] = int`
4. `[[(λ y:int => (x + y))]] = int -> int`
5. `[[λ x:int => (λ y:int => (x + y))]] = int -> [[(λ y:int => (x + y))]]`

Use constraint 4: replace the inner function's type by `int -> int` in constraint 5

Type Inference: Example 1 (12/13)

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

```
1.                                                [[x]] = int
2.                                                [[y]] = int
3.                                          [[(x + y)]] = int
4.                          [[(λ y:int => (x + y))]] = int -> int
5. [[λ x:int => (λ y:int => (x + y))]] = int -> (int -> int)
```

Use constraint 4: replace the inner function's type by `int -> int` in constraint 5

Type Inference: Example 1 (13/13)

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

1. `[[x]] = int`
2. `[[y]] = int`
3. `[[(x + y)]] = int`
4. `[[(λ y:int => (x + y))]] = int -> int`
5. `[[λ x:int => (λ y:int => (x + y))]] = int -> (int -> int)`

No contradiction: the program has type `int -> (int -> int)`

Type Inference: Example 2 (1/14)

```
42 (λ x:int => (x + 5))
```

Type Inference: Example 2 (2/14)

```
42 (λ x:int => (x + 5))
```

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

1. `[[42]] = int`

`[Int]` for 42 gives constraint 1

Type Inference: Example 2 (3/14)

```
42 (λ x:int => (x + 5))
```

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

1. `[[42]] = int`
2. `[[5]] = int`

`[Int]` for 5 gives constraint 2

Type Inference: Example 2 (4/14)

```
42 (λ x:int => (x + 5))
```

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

1. `[[42]] = int`
2. `[[5]] = int`
3. `[[x]] = int`
4. `[[(x + 5)]] = int`

`[Add]` for `x + 5` gives constraints 3 and 4

Type Inference: Example 2 (5/14)

```
42 (λ x:int => (x + 5))
```

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

```
1.                                    [[42]] = int
2.                                     [[5]] = int
3.                                     [[x]] = int
4.                                 [[(x + 5)]] = int
5.  [[(λ x:int => (x + 5))]] = [[x]] -> [[(x + 5)]]
```

[Def] for the function gives constraint 5

Type Inference: Example 2 (6/14)

```
42 (λ x:int => (x + 5))
```

$$\frac{\texttt{A} \vdash \texttt{e1 : t1 -> t2} \quad \texttt{A} \vdash \texttt{e2 : t1}}{\texttt{A} \vdash \texttt{e1 e2 : t2}}\ \texttt{[Call]}$$

```
1.                                          [[42]] = int
2.                                           [[5]] = int
3.                                           [[x]] = int
4.                                      [[(x + 5)]] = int
5.  [[(λ x:int => (x + 5))]] = [[x]] -> [[(x + 5)]]
6.              [[42]] = [[(λ x:int => (x + 5))]] ->
                         [[(42 (λ x:int => (x + 5)))]]
```

`[Call]` for the whole program gives constraint 6. Now let's solve the constraints . . .

Type Inference: Example 2 (7/14)

```
42 (λ x:int => (x + 5))
```

```
1.                                          [[42]] = int
2.                                           [[5]] = int
3.                                           [[x]] = int
4.                                     [[(x + 5)]] = int
5.  [[(λ x:int => (x + 5))]] = [[x]] -> [[(x + 5)]]
6.              [[42]] = [[(λ x:int => (x + 5))]] ->
                         [[(42 (λ x:int => (x + 5)))]]
```

Use constraint 1: replace `[[42]]` by `int` in constraint 6

Type Inference: Example 2 (8/14)

```
42 (λ x:int => (x + 5))
```

```
1.                                          [[42]] = int
2.                                           [[5]] = int
3.                                           [[x]] = int
4.                                    [[(x + 5)]] = int
5.  [[(λ x:int => (x + 5))]] = [[x]] -> [[(x + 5)]]
6.              int = [[(λ x:int => (x + 5))]] ->
                          [[(42 (λ x:int => (x + 5)))]]
```

Use constraint 1: replace `[[42]]` by `int` in constraint 6

Type Inference: Example 2 (9/14)

```
42 (λ x:int => (x + 5))
```

```
1.                                          [[42]] = int
2.                                           [[5]] = int
3.                                           [[x]] = int
4.                                     [[(x + 5)]] = int
5.  [[(λ x:int => (x + 5))]] = [[x]] -> [[(x + 5)]]
6.                int = [[(λ x:int => (x + 5))]] ->
                        [[(42 (λ x:int => (x + 5)))]]
```

Use constraint 3: replace `[[x]]` by `int` in constraint 5

Type Inference: Example 2 (10/14)

```
42 (λ x:int => (x + 5))
```

```
1.                                    [[42]] = int
2.                                     [[5]] = int
3.                                     [[x]] = int
4.                                [[(x + 5)]] = int
5.  [[(λ x:int => (x + 5))]] = int -> [[(x + 5)]]
6.              int = [[(λ x:int => (x + 5))]] ->
                      [[(42 (λ x:int => (x + 5)))]]
```

Use constraint 3: replace `[[x]]` by `int` in constraint 5

Type Inference: Example 2 (11/14)

```
42 (λ x:int => (x + 5))
```

```
1.                                    [[42]] = int
2.                                     [[5]] = int
3.                                     [[x]] = int
4.                                 [[(x + 5)]] = int
5.  [[(λ x:int => (x + 5))]] = int -> [[(x + 5)]]
6.              int = [[(λ x:int => (x + 5))]] ->
                          [[(42 (λ x:int => (x + 5)))]]
```

Use constraint 4: replace `[[(x + 5)]]` by `int` in constraint 5

Type Inference: Example 2 (12/14)

```
42 (λ x:int => (x + 5))
```

```
1.                                  [[42]] = int
2.                                   [[5]] = int
3.                                   [[x]] = int
4.                             [[(x + 5)]] = int
5.          [[(λ x:int => (x + 5))]] = int -> int
6.              int = [[(λ x:int => (x + 5))]] ->
                        [[(42 (λ x:int => (x + 5)))]]
```

Use constraint 4: replace `[[(x + 5)]]` by `int` in constraint 5

Type Inference: Example 2 (13/14)

```
42 (λ x:int => (x + 5))
```

1. `[[42]] = int`
2. `[[5]] = int`
3. `[[x]] = int`
4. `[[(x + 5)]] = int`
5. `[[(λ x:int => (x + 5))]] = int -> int`
6. `int = [[(λ x:int => (x + 5))]] -> [[(42 (λ x:int => (x + 5)))]]`

Use constraint 5: replace the function's type by `int -> int` in constraint 6

Type Inference: Example 2 (14/14)

`42 (λ x:int => (x + 5))` `int ≠ (int -> int) -> [[...]]` ✗

```
1.                                  [[42]] = int
2.                                   [[5]] = int
3.                                   [[x]] = int
4.                             [[(x + 5)]] = int
5.          [[(λ x:int => (x + 5))]] = int -> int
6.                          int = int -> int ->
                    [[(42 (λ x:int => (x + 5)))]]
```

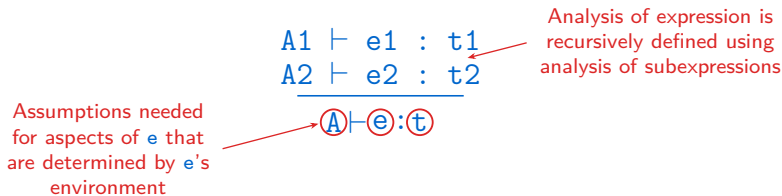
Constraint 6 asks `int` to equal a function type: unification fails (rule 4).

Static Analyses Using Type Rules (1/4)

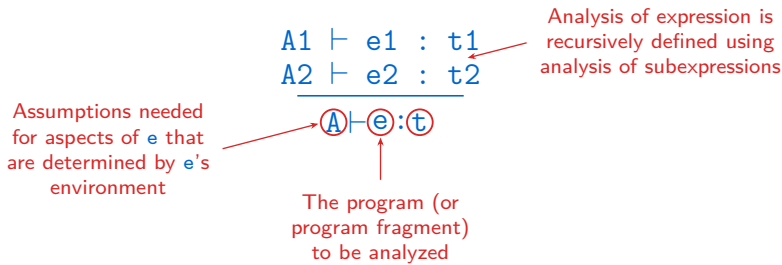
$$\frac{A1 \vdash e1 : t1 \qquad A2 \vdash e2 : t2}{A \vdash e : t}$$

Analysis of expression is recursively defined using analysis of subexpressions

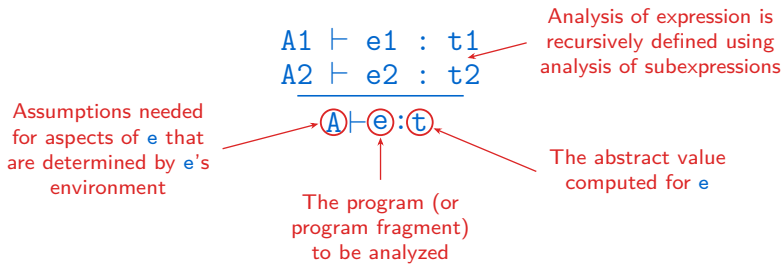
Static Analyses Using Type Rules (2/4)



Static Analyses Using Type Rules (3/4)



Static Analyses Using Type Rules (4/4)



Hypotheses may use other environments `A1`, `A2`, e.g. the extended `A[x ↦ t]` of `[Def]`.

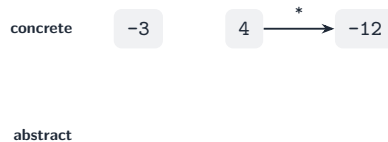
An Example: The Rule of Signs (1/3)

- Goal: to estimate the sign of a numeric computation



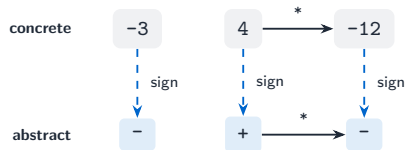
An Example: The Rule of Signs (2/3)

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



An Example: The Rule of Signs (3/3)

- Goal: to estimate the sign of a numeric computation
- Example: `-3 * 4 = -12`
- Abstraction: `- * + = -`
  ("negative times positive is negative")



Abstract Values (1/2)

- `+` = { all positive integers }
- `0` = {0}
- `-` = { all negative integers }

Abstract Values (2/2)

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

QUIZ: Example Rules, Part I (1/2)

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

$$\frac{A \vdash e1 : + \quad A \vdash e2 : -}{A \vdash e1 * e2 : \square}$$

$$\frac{A \vdash e1 : + \quad A \vdash e2 : +}{A \vdash e1 * e2 : \square}$$

$$\frac{A \vdash e1 : - \quad A \vdash e2 : -}{A \vdash e1 * e2 : \square}$$

$$\frac{A \vdash e1 : 0 \quad A \vdash e2 : +}{A \vdash e1 * e2 : \square}$$

QUIZ: Example Rules, Part I (2/2)

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

$$\frac{A \vdash e1 : + \quad A \vdash e2 : -}{A \vdash e1 * e2 : \boxed{-}}$$

$$\frac{A \vdash e1 : + \quad A \vdash e2 : +}{A \vdash e1 * e2 : \boxed{+}}$$

$$\frac{A \vdash e1 : - \quad A \vdash e2 : -}{A \vdash e1 * e2 : \boxed{+}}$$

$$\frac{A \vdash e1 : 0 \quad A \vdash e2 : +}{A \vdash e1 * e2 : \boxed{0}}$$

Same signs give +, different signs give −, and 0 times anything is 0.

Rule of Signs: A Derivation (1/3)

$$\texttt{A} \vdash \texttt{-3 * 4 : ?}$$

`[Int]` (axiom): $\texttt{A} \vdash \texttt{i : sign(i)}$, where `sign(i)` is `+`, `0` or `-`

Rule of Signs: A Derivation (2/3)

$$\frac{\texttt{A} \vdash \texttt{-3} : \texttt{-} \qquad \texttt{A} \vdash \texttt{4} : \texttt{+}}{\texttt{A} \vdash \texttt{-3 * 4} : \texttt{-}}\ \texttt{[Mul]}$$

`[Int]` (axiom): $\texttt{A} \vdash \texttt{i} : \texttt{sign(i)}$, where `sign(i)` is `+`, `0` or `-`

`[Mul]`: different signs give `-` (quiz)

Rule of Signs: A Derivation (3/3)

$$\dfrac{\dfrac{}{\texttt{A} \vdash \texttt{-3 : -}}\,[\texttt{Int}] \qquad \dfrac{}{\texttt{A} \vdash \texttt{4 : +}}\,[\texttt{Int}]}{\texttt{A} \vdash \texttt{-3 * 4 : -}}\,[\texttt{Mul}]$$

`[Int]` (axiom): $\texttt{A} \vdash \texttt{i : sign(i)}$, where `sign(i)` is `+`, `0` or `-`

`[Mul]`: different signs give `-` (quiz)

Proved: `-3 * 4` is **negative**, without computing `-12`

Revisiting Abstract Domains

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

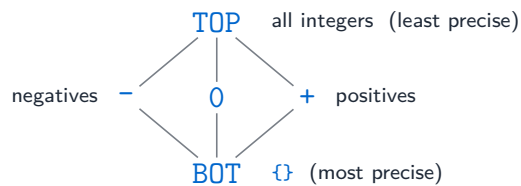
`3 + (-1) = 2`	positive
`1 + (-3) = -2`	negative
`2 + (-2) = 0`	zero

The sum can have **any** sign.

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

Solution:

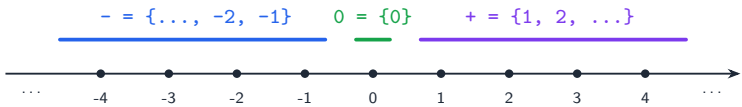
Add values so that every result is again an abstract value:



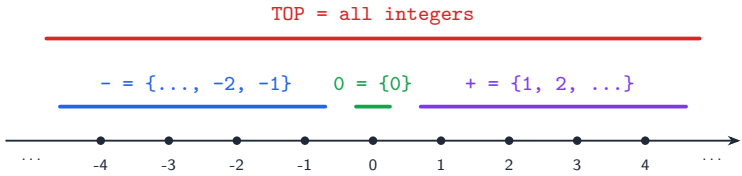
lower = subset; `+` plus `-` = least value above `+`, `0`, `-` = `TOP` (the join)

now `A : Variables -> {BOT, -, 0, +, TOP}`

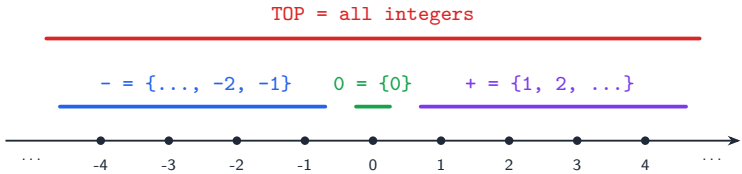
Abstract Values as Sets (1/5)



Abstract Values as Sets (2/5)

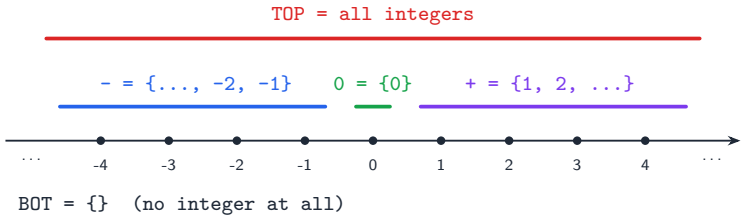


Abstract Values as Sets (3/5)

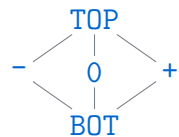


BOT = {} (no integer at all)

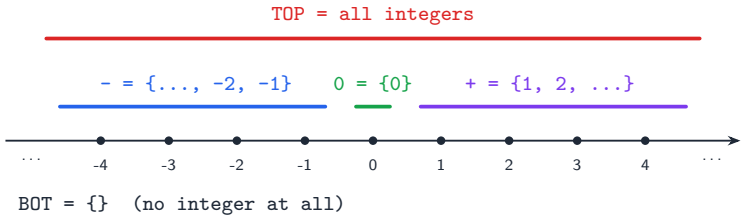
Abstract Values as Sets (4/5)



Order = subset: BOT $\subseteq$ - $\subseteq$ TOP



Abstract Values as Sets (5/5)



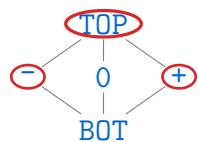
Order = subset: BOT $\subseteq$ - $\subseteq$ TOP

Join = smallest value containing both:

+ $\sqcup$ - = TOP + $\sqcup$ BOT = + - $\sqcup$ TOP = TOP

0 $\sqcup$ BOT = 0 0 $\sqcup$ TOP = TOP

in general: v $\sqcup$ BOT = v, v $\sqcup$ TOP = TOP



QUIZ: Example Rules, Part II (1/2)

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

$$\frac{\texttt{A} \vdash \texttt{e1} : + \quad \texttt{A} \vdash \texttt{e2} : -}{\texttt{A} \vdash \texttt{e1 + e2} : \square}$$

$$\frac{\texttt{A} \vdash \texttt{e1} : + \quad \texttt{A} \vdash \texttt{e2} : +}{\texttt{A} \vdash \texttt{e1 + e2} : \square}$$

$$\frac{\texttt{A} \vdash \texttt{e1} : 0 \quad \texttt{A} \vdash \texttt{e2} : +}{\texttt{A} \vdash \texttt{e1 / e2} : \square}$$

$$\frac{\texttt{A} \vdash \texttt{e1} : \texttt{TOP} \quad \texttt{A} \vdash \texttt{e2} : 0}{\texttt{A} \vdash \texttt{e1 / e2} : \square}$$

QUIZ: Example Rules, Part II (2/2)

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

$$\frac{A \vdash e1 : + \quad A \vdash e2 : -}{A \vdash e1 + e2 : \boxed{\texttt{TOP}}}$$

$$\frac{A \vdash e1 : + \quad A \vdash e2 : +}{A \vdash e1 + e2 : \boxed{+}}$$

$$\frac{A \vdash e1 : 0 \quad A \vdash e2 : +}{A \vdash e1 \,/\, e2 : \boxed{0}}$$

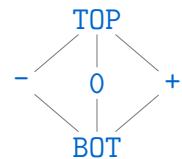
$$\frac{A \vdash e1 : \texttt{TOP} \quad A \vdash e2 : 0}{A \vdash e1 \,/\, e2 : \boxed{\texttt{BOT}}}$$

`+` plus `-` can have any sign: `TOP`. Division by `0` has no result: `BOT`.

Using TOP and BOT: An Example (1/5)

Environment `A = [x ↦ +, y ↦ -, z ↦ 0]`

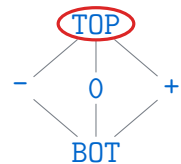
Expression	As sets of integers	Value



Using TOP and BOT: An Example (2/5)

Environment `A = [x ↦ +, y ↦ -, z ↦ 0]`

Expression	As sets of integers	Value
`x + y`	$\{ p + n : p > 0,\ n < 0 \}$ = all integers	**TOP**

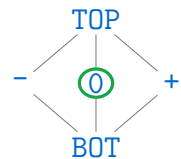


circle: the value of
the current expression

Using TOP and BOT: An Example (3/5)

Environment `A = [x` $\mapsto$ `+,  y` $\mapsto$ `-,  z` $\mapsto$ `0]`

Expression	As sets of integers	Value
`x + y`	$\{ p + n : p > 0,\ n < 0 \} =$ all integers	**TOP**
`(x + y) * z`	$\{ k \cdot 0 : k$ any integer $\} = \{0\}$	**0**

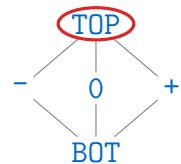


circle: the value of
the current expression

Using TOP and BOT: An Example (4/5)

Environment `A = [x ↦ +, y ↦ -, z ↦ 0]`

Expression	As sets of integers	Value
`x + y`	$\{ p + n : p > 0,\ n < 0 \}$ = all integers	**TOP**
`(x + y) * z`	$\{ k \cdot 0 : k$ any integer$\} = \{0\}$	**0**
`(x + y) * x`	$\{ k \cdot p : k$ any, $p > 0 \}$ = all integers	**TOP**

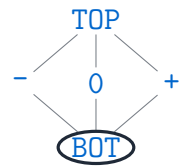


circle: the value of the current expression

Using TOP and BOT: An Example (5/5)

Environment `A = [x ↦ +, y ↦ -, z ↦ 0]`

Expression	As sets of integers	Value
`x + y`	$\{ p + n : p > 0,\ n < 0 \}$ = all integers	**TOP**
`(x + y) * z`	$\{ k \cdot 0 : k$ any integer $\} = \{0\}$	**0**
`(x + y) * x`	$\{ k \cdot p : k$ any, $p > 0 \}$ = all integers	**TOP**
`x / z`	$\{ p/0 : p > 0 \} = \{\}$ (no result)	BOT



circle: the value of the current expression

Flow Insensitivity (1/3)

$$\frac{\begin{array}{l}\texttt{A} \vdash \texttt{e0 : bool} \\ \quad \texttt{A} \vdash \texttt{e1 : t1} \\ \quad \texttt{A} \vdash \texttt{e2 : t2} \\ \quad\quad \texttt{t1 = t2}\end{array}}{\texttt{A} \vdash \texttt{if e0 then e1 else e2 : t1}}$$

Subexpressions are independent of each other

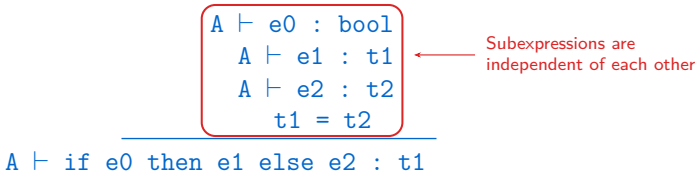
Flow Insensitivity (2/3)

$$\frac{\begin{array}{l}\texttt{A} \vdash \texttt{e0 : bool} \\ \quad \texttt{A} \vdash \texttt{e1 : t1} \\ \quad \texttt{A} \vdash \texttt{e2 : t2} \\ \quad\quad \texttt{t1 = t2}\end{array}}{\texttt{A} \vdash \texttt{if e0 then e1 else e2 : t1}}$$

Subexpressions are independent of each other

- **Flow-insensitive analysis**: analysis is independent of ordering of sub-expressions
- $\Rightarrow$ analysis result unaffected by permuting statements

Flow Insensitivity (3/3)



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

Flow Insensitivity (1/3)

- Flow insensitive analyses are often very **efficient and scalable**

Flow Insensitivity (2/3)

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

Flow Insensitivity (3/3)

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

Flow Sensitivity (1/3)

$$\frac{\begin{array}{l} \texttt{A} \vdash \texttt{e0 : bool} \triangleright \texttt{A0} \\ \texttt{A0} \vdash \texttt{e1 : t1} \triangleright \texttt{A1} \\ \texttt{A0} \vdash \texttt{e2 : t2} \triangleright \texttt{A2} \\ \texttt{t1 = t2,} \quad \texttt{A1 = A2} \end{array}}{\texttt{A} \vdash \texttt{if e0 then e1 else e2 : t1} \triangleright \texttt{A1}}$$

Rules produce new environments, and analysis of a subexpression cannot happen until its environment is available

`A ⊢ e : t ▷ A'`: under `A`, `e` has type `t` and leaves the environment `A'`

Flow Sensitivity (2/3)

$$\frac{\begin{array}{l} A \vdash e0 : bool \triangleright A0 \\ A0 \vdash e1 : t1 \triangleright A1 \\ A0 \vdash e2 : t2 \triangleright A2 \\ t1 = t2, \quad A1 = A2 \end{array}}{A \vdash \texttt{if } e0 \texttt{ then } e1 \texttt{ else } e2 : t1 \triangleright A1}$$

Rules produce new environments, and analysis of a subexpression cannot happen until its environment is available

`A ⊢ e : t ▷ A'`: under `A`, `e` has type `t` and leaves the environment `A'`

- **Flow-sensitive analysis**: analysis of subexpressions ordered by environments
- $\Rightarrow$ analysis result depends on **order of statements**

Flow Sensitivity (3/3)

$$\frac{\begin{array}{l} A \vdash e0 : bool \triangleright A0 \\ A0 \vdash e1 : t1 \triangleright A1 \\ A0 \vdash e2 : t2 \triangleright A2 \\ t1 = t2, \quad A1 = A2 \end{array}}{A \vdash \texttt{if e0 then e1 else e2} : t1 \triangleright A1}$$

Rules produce new environments, and analysis of a subexpression cannot happen until its environment is available

`A ⊢ e : t ▷ A'`: under `A`, `e` has type `t` and leaves the environment `A'`

- **Flow-sensitive analysis**: analysis of subexpressions ordered by environments
- $\Rightarrow$ analysis result depends on **order of statements**
- **Dataflow analysis** is an example of a flow-sensitive analysis

Flow Sensitivity

- Flow-sensitive analysis can be expensive

  Each statement has own **model of state**

  Polynomial cost increase over flow-insensitive

  (one environment per program point instead of one in total)

Path Sensitivity (1/4)

$$\frac{\begin{array}{l}\texttt{P, A} \vdash \texttt{e0 : bool} \triangleright \texttt{A0} \\ \texttt{P} \wedge \texttt{e0, A0} \vdash \texttt{e1 : t1} \triangleright \texttt{A1} \\ \texttt{P} \wedge \neg\texttt{e0, A0} \vdash \texttt{e2 : t2} \triangleright \texttt{A2} \\ \texttt{t1 = t2}\end{array}}{\texttt{P, A} \vdash \texttt{if e0 then e1 else e2 : t1, e0 ? A1 : A2}}$$

Path Sensitivity (2/4)

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

$$\frac{\begin{array}{l} \texttt{P, A} \vdash \texttt{e0 : bool} \triangleright \texttt{A0} \\ \texttt{P} \wedge \texttt{e0, A0} \vdash \texttt{e1 : t1} \triangleright \texttt{A1} \\ \texttt{P} \wedge \neg\texttt{e0, A0} \vdash \texttt{e2 : t2} \triangleright \texttt{A2} \\ \texttt{t1 = t2} \end{array}}{\texttt{P, A} \vdash \texttt{if e0 then e1 else e2 : t1, e0 ? A1 : A2}}$$

Path Sensitivity (3/4)

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

$$\frac{\begin{array}{rl} \texttt{P}, & \texttt{A} \vdash \texttt{e0} : \texttt{bool} \triangleright \texttt{A0} \\ \texttt{P} \wedge & \texttt{e0}, \texttt{A0} \vdash \texttt{e1} : \texttt{t1} \triangleright \texttt{A1} \\ \texttt{P} \wedge \neg & \texttt{e0}, \texttt{A0} \vdash \texttt{e2} : \texttt{t2} \triangleright \texttt{A2} \\ & \texttt{t1} = \texttt{t2} \end{array}}{\texttt{P}, \texttt{A} \vdash \texttt{if e0 then e1 else e2} : \texttt{t1}, \texttt{e0} \; ? \; \texttt{A1} : \texttt{A2}}$$

Part of the environment is a predicate saying under what condition this expression is executed

Path Sensitivity (4/4)

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

$$\frac{\begin{array}{rl} \texttt{P}, & \texttt{A} \vdash \texttt{e0} : \texttt{bool} \triangleright \texttt{A0} \\ \texttt{P} \wedge & \texttt{e0}, \texttt{A0} \vdash \texttt{e1} : \texttt{t1} \triangleright \texttt{A1} \\ \texttt{P} \wedge \neg & \texttt{e0}, \texttt{A0} \vdash \texttt{e2} : \texttt{t2} \triangleright \texttt{A2} \\ & \texttt{t1} = \texttt{t2} \end{array}}{\texttt{P}, \texttt{A} \vdash \texttt{if e0 then e1 else e2} : \texttt{t1}, \texttt{e0 ? A1 : A2}}$$

Part of the environment is a predicate saying under what condition this expression is executed

At points where control paths merge, still keep different paths separate in the final environment

`e0 ? A1 : A2`: the output is `A1` on paths where `e0` holds, `A2` otherwise

Comments on Path Sensitivity (1/3)

- **Symbolic execution**[1] is an example

  Path-sensitive analyses are also flow-sensitive

[1] Symbolic execution: runs the program on symbols instead of concrete inputs, and records for each path the condition on the inputs under which that path is taken.

Comments on Path Sensitivity (2/3)

- **Symbolic execution**[1] is an example

  Path-sensitive analyses are also flow-sensitive

- Can be expensive but a necessary evil

  Exponential number of paths to track

[1] Symbolic execution: runs the program on symbols instead of concrete inputs, and records for each path the condition on the inputs under which that path is taken.

Comments on Path Sensitivity (3/3)

- **Symbolic execution**[1] is an example

  Path-sensitive analyses are 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

[1] Symbolic execution: runs the program on symbols instead of concrete inputs, and records for each path the condition on the inputs under which that path is taken.

QUIZ: Flow and Path Sensitivity (1/2)

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

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

QUIZ: Flow and Path Sensitivity (2/2)

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

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

Summary

Type systems = **flow-insensitive**

Dataflow analysis = **flow-sensitive**

Symbolic execution = **path-sensitive**