Scala

Lecture 6b onwards

This markdown file documents everything learned in 50.054 Compiler Design and Program Analysis related to Scala

Pseudo Assembly (6B)

back-to-top

Recall the compiler pipeline

- `Source Text ⟶ [Lexing] ⟶ [Parsing] ⟶` **`[Semantic Analysis]`** `⟶ [Optimization] ⟶ [Code Generation]`

SIMP (Simple Imperative Programming language)

back-to-top

- Tiny imperative language with assignments

Syntax

```
Statements (S):
  S ::= X = E ;
      | return X ;
      | nop ;
      | if E { S } else { S }
      | while E { S }
      | S S          // sequencing (two statements in a row)

Expressions (E):
  E ::= E OP E
      | X
      | C
      | ( E )

Operators (OP):
  OP ::= + | - | * | < | ==

Constants (C):
  C ::= 0 | 1 | 2 | ... | true | false

Variables (X):
  X ::= a | b | c | d | ...
```

Example

```
x = input;
s = 0;
c = 0;
while c < x {
  s = c + s;
}
return s
```

Pseudo Assembly Syntax

back-to-top

(Instruction) i ::= $d \leftarrow s$ `(move)` | $d \leftarrow s \; op \; s$ `(binary)` | *ret* `return` | *ifn s goto l* `(if not s go to l)` | *goto l* `(unconditional jump)`

(Operand) *d,s* ::= r | c | t
(Temp Var) *t* ::= x | y | ...
(Label) *l* ::= 1 | 2 |...
(Operator) *op* ::= + | - | * |...
(Constant) *c* ::= 0 | 1 | 2 | ... `All natural numbers`
(Register) *r* ::= rret | r1 | r2 `rret: return register, stores the return statement`

Example

```
1: x <- input
2: s <- 0
3: c <- 0
4: t <- c < x
5: ifn t goto 9
6: s <- c + s
7: c <- c + 1
8: goto 4
9: rret <- s
10: ret
```

Maximal Munch (SIMP to PA)

back-to-top

Maximal Munch → rules that tell you how to generate code

Always grab the biggest SIMP chunk you can turn into PA in one go

Converting SIMP to Pseudo Assembly

```
SIMP                                          Psuedo Assembly
#############                                 ################
x = input;                                    1: x <- input
s = 0;                                        2: s <- 0
c = 0;                                        3: c <- 0
while c < x {         -> Maximal Munch ->     4: t <- c < x
  s = c + s;                                  5: ifn t goto 9
  c = c + 1;                                  6: s <- c + s
}                                             7: c <- c + 1
return s;                                     8: goto 4
                                              9: rret <- s
                                              10: ret
```

How? Two mutually defined judgments:

- Statements: use `G_s(S) ⊢ lis` to turn a SIMP statement into labeled PA instructions.
- Expressions: use `G_a(X)(E) ⊢ lis` to compute an expression into destination `X`.

Assignment rule:

$$\frac{G_a(X)(E) \vdash \text{lis}}{G_s(X = E) \vdash \text{lis}} \quad (\text{mAssign})$$

Constant rule (with fresh label l):

$$\frac{c = \text{conv}(C)}{G_a(X)(C) \vdash [\, l : X \leftarrow c \,]} \quad (\text{mConst})$$

where

$$\text{conv}(\text{true}) = 1, \quad \text{conv}(\text{false}) = 0, \quad \text{conv}(C) = C.$$

Variable rule:

$$\frac{l \text{ fresh}}{G_a(X)(Y) \vdash [\, l : X \leftarrow Y \,]} \quad (\text{mVar})$$

Binary-operator rule:

$$\frac{\begin{array}{c} t_1 \text{ fresh},\ G_a(t_1)(E_1) \vdash \text{lis}_1 \\ t_2 \text{ fresh},\ G_a(t_2)(E_2) \vdash \text{lis}_2 \\ l \text{ fresh} \end{array}}{G_a(X)(E_1 \text{ OP } E_2) \vdash \text{lis}_1 + \text{lis}_2 + [\, l : X \leftarrow t_1 \text{ OP } t_2 \,]} \quad (\text{mOp})$$

Parentheses rule:

$$\frac{G_a(X)(E) \vdash \text{lis}}{G_a(X)((E)) \vdash \text{lis}} \quad (\text{mParen})$$

Return rule:

$$\frac{G_a(r_{ret})(X) \vdash \text{lis} \quad l \text{ fresh}}{G_s(\text{return } X) \vdash \text{lis} + [\, l : \text{ret} \,]} \quad (\text{mReturn})$$

Sequence rule:

$$\frac{\forall i \in \{1, \ldots, n\},\ G_s(S_i) \vdash \text{lis}_i}{G_s(S_1; \ldots ; S_n) \vdash \text{lis}_1 + \cdots + \text{lis}_n} \quad (\text{mSequence})$$

No-op rule:

$$G_s(\text{nop}) \vdash [\,] \quad (\text{mNOp})$$

- `mAssign` : compile the expression with `G_a(X)(E)` so its value lands in `X` , then reuse that instruction list for the statement `X = E` .
- `mConst` : when the expression is a literal `C` , emit a fresh instruction `l: X <- conv(C)` moving the constant directly into `X` .
- `mReturn` : evaluate `X` into the special register `r_ret` , then append a fresh `ret` so returning statements terminate with that value.
- `mSequence` : translate each statement `S_i` in order and concatenate their instruction lists to preserve sequential execution.
- `mNOp` : do nothing— `nop` deliberately produces an empty instruction list.
- `mVar` : load a variable `Y` into destination `X` by emitting a single move instruction with a fresh label.
- `mOp` : evaluate both operands into temporary destinations, then emit an instruction that combines them with the operator and stores the result in `X` .
- `mParen` : parentheses don't change code generation, so just reuse the instructions produced for the inner expression `E` .

Maximal Munch Example Deep Dive

back-to-top

Converting SIMP to Pseudo Assembly

```
SIMP                                            Psuedo Assembly
#############                                   ################
x = input;                                      1: x <- input
s = 0;                                          2: s <- 0
c = 0;                                          3: c <- 0
while c < x {          -> Maximal Munch ->      4: t <- c < x
  s = c + s;                                    5: ifn t goto 9
  c = c + 1;                                    6: s <- c + s
}                                               7: c <- c + 1
return s;                                       8: goto 4
                                                9: rret <- s
                                                10: ret
```

Mapping SIMP → PA with rule invocations:

SIMP fragment	PA line(s)	Rule invocation	Key details
`x = input;`	`1: x <- input`	`G_s(x = input)` uses `mAssign` ; expression side runs `G_a(x)(input)`	`input` expression already matches a PA move; result stored directly in destination `x` .
`s = 0;`	`2: s <- 0`	`G_a(s)(0)` hits `mConst` inside `mAssign`	`conv(0) = 0` , so the constant rule emits a fresh labeled instruction moving literal 0 into `s` .
`c = 0;`	`3: c <- 0`	`G_a(c)(0)` with `mConst`	Same `conv(0) = 0` behavior; fresh label for `c` .
`while c < x { … }` guard eval	`4: t <- c < x`	`G_s(while …)` emits guard using `G_a(t)(c < x)` with `mOp` + `mVar`	Guard label `4` is "fresh" so the loop can jump back; `t` stores comparison result.
`while` guard branch	`5: ifn t goto 9`	`G_s(while …)` loop rule	Conditional branch exits loop when guard fails; target label `9` is preallocated for the loop exit.
`s = c + s;` (body)	`6: s <- c + s`	`G_a(s)(c + s)` via `mAssign` + `mOp` + `mVar`	Binary-op rule evaluates operands, then emits instruction storing `c + s` back to `s` .
`c = c + 1;` (body)	`7: c <- c + 1`	`G_a(c)(c + 1)` using `mOp` + `mConst`	Uses fresh temps plus `conv(1)=1` to increment `c` ; still part of loop body sequence.

SIMP fragment	PA line(s)	Rule invocation	Key details
loop back-edge	`8: goto 4`	`G_s(while …)` loop tail	Implements the back-edge: unconditional jump to guard label `4` to re-test the loop condition.
`return s;` (move)	`9: rret <- s`	`G_s(return s)` invokes `mReturn` with `G_a(r_ret)(s)` (`mVar`)	Return rule first moves `s` into `r_ret`; this is analogous to `G_a(r_ret)(s)` using `mVar`.
`return s;` (ret)	`10: ret`	`mReturn`	Final `ret` instruction (with fresh label) completes the return statement.

1. `x = input`

- Treat as an assignment statement
- Applies `(mAssign)` : call `G_a(x)(input)` to emit code that computes `input` into dest `x`
- *When G_a(X)(E) finishes, the computed value of expression E must be stored in X. So X is simply the place you've chosen to hold the result*
- The expression `input` already corresponds to a PA instruction, so the result is `1: x <- input`

2. `s = 0`

- Applies `(mAssign)` : call `G_a(s)(0)`
- Constant `0` hits `(mConst)` with fresh label `2` and `conv(0) = 0`
- *conv(0) = 0 means if the SIMP constant is 0, the instruction should write 0 into the destination.*
- Emit `2: s <- 0`

3. `c = 0`

- Applies `(mAssign)` : call `G_a(c)(0)`
- Constant `0` hits `(mConst)` with fresh label `3` and `conv(0) = 0`
- Emit `3: s <- 0`

4. `while c < x { body }`

- `G_s` sees a loop, so it emits a label for the guard
- *When compiling a while loop, the guard (the Boolean test) needs a jump target so execution can return to it after each iteration. "Emit a label for the guard" means: insert a numbered label on the PA line where the guard expression is evaluated (e.g., 4:). Later, after the loop body, the generated code includes goto 4, sending control back to that labeled guard line to re-test the condition.*
- Guard: `4: t <- c < x` → the thing that terminates the while loop basically
- Conditional jump: `5: ifn t goto 9` (exit label)

5. Loop body statement `s = c + s`

- `(mAssign)` with expression `c + s`
- `G_a(s)(c + s)` emits `6: s <- c + s`

6. Next body statement `c = c + 1`

- `(mAssign)` with expression `c + 1`
- `G_a(c)(c + 1)` emits `7: c <- c + 1`

7. Loop Tail

- After the body, `G_s` emits a goto back to the guard

- emits `8: goto 4`

8. After loop, `return s`

- Return becomes move in `rret` plus `ret`
- emits `9: rret < - s` and `10: ret`

Issues with Maximal Munch

back-to-top

- too many temp variables

Example

```
z = x - (y + 1) --> (apply mAssign) --> G_a(z)(x - (y + 1))

3: t1 <- x,
4: t3 <- y,
5: t4 <- 1,
6: t2 <- t3 + t4 (y+1)
7: z  <- t1 - t2 (x- (y+1))
```

There is too many temp var like t1, t2, t3, t4

Optimised Encoding

```
3: t1 <- y + 1
4: t2 <- x - t1
5: z  <- t2
```

Semantic Analysis (8A)

- `Source Text → [Lexing] → [Parsing] → `**`[Semantic Analysis]`**` → [Optimization] → [Code Generation]`

back-to-top

- Syntax Analysis → verify given code conforms to grammar rules
- Semantic Analysis → verify code behave according to human expectation → how does the program get executed, what does the program compute/return?
  - Goal #1: Optimization
  - Goal #2: Fault Detection

Dynamic Semantic Analysis

back-to-top

The program's behavior when it *runs*. How does it *execute*? What does it *compute/return*?
Approaches:

1. Testing

2. Run-time verification → finding bugs by log
3. Fuzzing

Static Semantics Analysis

back-to-top

What the program MUST satisfy BEFORE excution

Approaches:

1. Type rules checking
2. Name rules analysis
3. Flow rules

- **Type checking**

```
b = true;
c = b + 1
// Static error: Type mismatch
```

- **Name analysis** → is it a function?

```
y = x + 1;
return y;
// Static error: x is undefined
```

- **Control flow analysis** →“which statements can execute?”, “where are loops/back-edges?”, and “is code unreachable?”.
- **Data flow analysis** → how data flow
- **Model checking** → step by step input-output checking
- **Abstract intepretation** → Approximate program behavior using simpler values to prove properties.
- **Symbolic Execution**

Goal #1: Fault Detection

Divide by 0 math error

back-to-top

```
x = input;
while (x>=0){
    x = x - 1
}
y = Math.sqrt(x)
return y;

// Will raise error because if x = 0, you will be square rooting -1
```

```
1: x <- input
2: t1 <- 0 < x
3: t2 <- 0 == x
4: t3 <- t1 + t2 // t1 or t2
5: ifn t3 goto 8
6: x = x - 1
7: goto 2
8: y <- Math.sqrt(x) // x is definitely negative
9: rret <- y
10: ret
```

Goal #2: Optimization

back-to-top

```
x = input;
y = 0;
s = 0;
while (y < x) {
    y = y + 1;
    t = s; // t is not used.
    s = s + y;
}
return s;
```

```
1: x <- input
2: y <- 0
3: s <- 0
4: b <- y < x
5: ifn b goto 10
6: y <- y + 1
7: t <- s // t is not used
8: s <- s + y
9: goto 4
10: rret <- s
11: ret
```

Rice Theorem

back-to-top

All non-trivial semantic properties of programs are undecidable, i.e. there exists no algorithm that can decide all semantic properties fro all given programs

Example: There exists **NO** algorithm in the world that can determine if the x is 1 or -1 without running it

→ the Halting Problem

```
def f(path):
    p = open(path, "r")
    x = 1
    if eval(p):
        x = -1
    return x
```

- `eval()` → evaluates a python expression and returns its value

```
eval("2 + 3")
#5

f = eval("lambda x: x*x")
f(3)
# 9
```

Dynamic Semantics (8B)

back-to-top

A formal way to specify a language's run-time behavior: how programs execute and what they return

Types of Dynamic Semantics

1. **Operational Semantics**

- Define execution as **term rewriting** (rules that transform program states)
- Small-step → one micro step at a time until you reach a normal form (or continue forever)
- Big-step → jump directly to results

2. **Denotational Semantics**

- Map programs to mathematical objects (meanings)

3. **Translational Semantics**

- Translate one language to another

Small Step (Operational Semantics)

back-to-top

Good for debugger mode: shows every intermediate configurations. It's for giving an unambiguous "official recipe" for how a program runs, one tiny step at a time

A small-step semantics is a relation (often written as: →) defined by **inference rules**

Inference rules defines the small-steps:

$$\frac{\text{premises}}{\text{conclusion}}$$

Meaning: if the `premises` is true, you may apply the rule to conclude the bottom `conclusion` (one step done)

All SIMP Small Step inference rules:

back-to-top

$$\textbf{(sVar)} \Delta \vdash X \longrightarrow \Delta(X)$$

Meaning: look up variable `X` in the store and replace it with its value.

$$\textbf{(sOp1)} \quad \frac{\Delta \vdash E_1 \longrightarrow E_1'}{\Delta \vdash E_1 \text{ OP } E_2 \longrightarrow E_1' \text{ OP } E_2}$$

Meaning: take one evaluation step in the left operand of a binary operation.

$$\textbf{(sOp2)} \quad \frac{\Delta \vdash E_2 \longrightarrow E_2'}{\Delta \vdash C_1 \text{ OP } E_2 \longrightarrow C_1 \text{ OP } E_2'}$$

Meaning: when the left operand is already a constant, evaluate the right operand next.

$$\textbf{(sOp3)} \quad \frac{C_3 = C_1 \text{ OP } C_2}{\Delta \vdash C_1 \text{ OP } C_2 \longrightarrow C_3}$$

Meaning: if both operands are constants, compute the primitive result immediately.

$$\textbf{(sParen1)} \quad \frac{\Delta \vdash E \longrightarrow E'}{\Delta \vdash (E) \longrightarrow (E')}$$

Meaning: evaluate inside parentheses (propagate a step into the grouped expression).

$$\textbf{(sParen2)} \quad \Delta \vdash (c) \longrightarrow c$$

Meaning: parentheses around a constant disappear — the constant is the result.

$$\textbf{(sAssign1)} \quad \frac{\Delta \vdash E \longrightarrow E'}{(\Delta,\ X = E\ ;) \longrightarrow (\Delta,\ X = E'\ ;)}$$

Meaning: evaluate the right-hand side expression before performing the assignment.

$$\textbf{(sAssign2)} \quad \frac{\Delta' = \Delta \oplus (X, C)}{(\Delta,\ X = C\ ;) \longrightarrow (\Delta',\ \text{nop})}$$

Meaning: when RHS is a constant, update the store with `X = C` and replace the statement with `nop`.

$$\textbf{(sIf1)} \quad \frac{\Delta \vdash E \longrightarrow E'}{(\Delta,\ \text{if } E\{S_1\} \text{ else } \{S_2\}) \longrightarrow (\Delta,\ \text{if } E'\{S_1\} \text{ else } \{S_2\})}$$

Meaning: evaluate the `if` condition expression before choosing a branch.

$$(\textbf{sIf2})(\Delta,\ \text{if true}\ \{S_1\}\ \text{else}\ \{S_2\}) \longrightarrow (\Delta,\ S_1)$$

Meaning: if the condition is `true`, take the then-branch `S_1`.

$$(\textbf{sIf3})(\Delta,\ \text{if false}\ \{S_1\}\ \text{else}\ \{S_2\}) \longrightarrow (\Delta,\ S_2)$$

Meaning: if the condition is `false`, take the else-branch `S_2`.

$$(\textbf{sWhile})\big(\Delta,\ \text{while}\ E\ \{S\} \longrightarrow (\Delta,\ \text{if}\ E\ \{\ S;\ \text{while}\ E\ \{S\}\ \}\ \text{else}\ \{\text{nop}\})\big)$$

Meaning: unroll the `while` loop as an equivalent `if` that repeats the body when the guard holds.

$$(\textbf{sWhile1})\quad \frac{\Delta \vdash E \longrightarrow E'}{(\Delta,\ \text{while}\ E\{S\}) \longrightarrow (\Delta,\ \text{while}\ E'\{S\})}$$

Meaning: take a step to evaluate the loop condition expression.

$$(\textbf{sWhile2})(\Delta,\ \text{while true}\ \{S\}) \longrightarrow (\Delta,\ S\ ;\ \text{while true}\ \{S\})$$

Meaning: if the guard is `true`, execute the body `S` then loop again.

$$(\textbf{sWhile3})(\Delta,\ \text{while false}\ \{S\}) \longrightarrow (\Delta,\ \text{nop})$$

Meaning: if the guard is `false`, exit the loop (do nothing).

$$(\textbf{sNopSeq})(\Delta,\ \text{nop};\bar{S}) \longrightarrow (\Delta,\ \bar{S})$$

Meaning: drop a leading `nop` and continue with the remaining sequence.

$$(\textbf{sSeq})\quad \frac{S \neq \text{nop} \quad (\Delta,\ S) \longrightarrow (\Delta',\ S')}{(\Delta,\ S; SS) \longrightarrow (\Delta',\ S'; SS)}$$

Meaning: evaluate the left statement `S` first; when it steps to `S'` keep the sequence order.

Legend (symbols & notation)

back-to-top

Symbol	Meaning
Δ	Store / environment mapping variables to constants (memory)
⊢ (turnstile)	Judgment turnstile; used to relate a store and an expression/statement to a transition
→	One small-step transition (single computation step)
→*	Multi-step (reflexive-transitive) closure of → (zero or more steps)

Symbol	Meaning
overline / axiom (e.g. `\overline{(... )}`)	A rule with no premises (an axiom) — shows the conclusion directly
`nop`	No-op statement (does nothing)
(Δ, S)	Configuration: store Δ together with statement `S` (program state)
C	Constant literal (e.g., 0, true, false)
X	Variable name
E, E'	Expressions (possibly reducible)

Notes:

- When a rule uses a condition like `C_3 = C_1 OP C_2` , it means the result of applying `OP` to the constants `C_1` and `C_2` produces `C_3` (a primitive computation).
- Sequencing `S ; SS` means `S` followed by the rest `SS` (the semantics reduce `S` until `nop` , then continue with `SS`).

Example of Small Step with SIMP

back-to-top

Goal: with small-step we take the SIMP program and turn it into a chain like:

$(\Delta_0, S_0) \to (\Delta_1, S_1) \to (\Delta_2, S_2) \to \ldots$

where each arrow is one rule application (one legal next move)

Now given this example below, we want to formally justify the execution story:

```
x = input;
x = x + 1;
return x;
```

- when the `return` statement carries a constant value we consider the program terminated with value $v + 1$ (sometimes modeled by a special halt configuration or by setting a return register).
- Final store: Δ_2 with $x \mapsto v + 1$.
- Program result: value $v + 1$.

Compact chain (configuration sequence):

```
(Δ₀, x = input; x = x + 1; return x)
→ (Δ₀, x = v; x = x + 1; return x)         (sAssign1) — evaluate RHS `input` to value `v`
→ (Δ₁, nop; x = x + 1; return x)           (sAssign2) — update store `x ↦ v` and replace assignment with `nop`
→ (Δ₁, x = x + 1; return x)                (sNopSeq)  — drop leading `nop` and continue
→ (Δ₁, x = v+1; return x)                  (sAssign1) — evaluate RHS `x + 1` to constant `v+1` (via sVar, sOp1, sOp3)
→ (Δ₂, nop; return x)                      (sAssign2) — update store `x ↦ v+1` and replace assignment with `nop`
→ (Δ₂, return x)                           (sNopSeq)  — drop leading `nop` and continue
→ halt with result v+1                     (return evaluation / halt) — program terminates with value `v+1`
```

Big Step (Operational Semantics)

back-to-top

All SIMP Big-step inference rules:

back-to-top

$$\textbf{(bConst)} \qquad \Delta \vdash C \Downarrow C$$

Meaning: a literal constant evaluates to itself.

$$\textbf{(bVar)} \qquad \Delta \vdash X \Downarrow \Delta(X)$$

Meaning: look up a variable `X` in the store and return its value.

$$\textbf{(bOp)} \quad \frac{\Delta \vdash E_1 \Downarrow C_1 \quad \Delta \vdash E_2 \Downarrow C_2 \quad C_3 = C_1 \text{ OP } C_2}{\Delta \vdash E_1 \text{ OP } E_2 \Downarrow C_3}$$

Meaning: evaluate both operands to constants then compute the primitive operator result.

$$\textbf{(bParen)} \quad \frac{\Delta \vdash E \Downarrow C}{\Delta \vdash (E) \Downarrow C}$$

Meaning: parentheses do not change the evaluated result (evaluate inner expression).

$$\textbf{(bAssign)} \quad \frac{\Delta \vdash E \Downarrow C}{(\Delta,\ X = E) \Downarrow (\Delta \oplus (X \mapsto C),\ \text{no-return})}$$

Meaning: evaluate the RHS to a constant, update the store with `X = C`, assignment completes in one big step.

$$\textbf{(bIf1)} \quad \frac{\Delta \vdash E \Downarrow \text{true} \quad (\Delta, S_1) \Downarrow (\Delta', r)}{(\Delta,\ \text{if } E\{S_1\} \text{ else } \{S_2\}) \Downarrow (\Delta', r)}$$

Meaning: if the guard evaluates to true, evaluate the `then` branch to completion.

$$\textbf{(bIf2)} \quad \frac{\Delta \vdash E \Downarrow \text{false} \quad (\Delta, S_2) \Downarrow (\Delta', r)}{(\Delta,\ \text{if } E\{S_1\} \text{ else } \{S_2\}) \Downarrow (\Delta', r)}$$

Meaning: if the guard evaluates to false, evaluate the `else` branch to completion.

$$\textbf{(bWhile1)} \quad \frac{\Delta \vdash E \Downarrow \text{true} \quad (\Delta, S) \Downarrow (\Delta_1,\ \text{no-return}) \quad (\Delta_1,\ \text{while } E\{S\}) \Downarrow (\Delta_2, r)}{(\Delta,\ \text{while } E\{S\}) \Downarrow (\Delta_2, r)}$$

Meaning: when the guard is true, execute the body then repeat the loop (big-step unrolling).

$$\textbf{(bWhile2)} \quad \frac{\Delta \vdash E \Downarrow \text{false}}{(\Delta,\ \text{while } E\{S\}) \Downarrow (\Delta,\ \text{no-return})}$$

Meaning: when the guard is false the loop terminates immediately (no effect on the store).

$$\textbf{(bNop)} \quad \frac{}{(\Delta,\ \text{nop}) \Downarrow (\Delta,\ \text{no-return})}$$

Meaning: `nop` does nothing and leaves the store unchanged.

$$\textbf{(bSeq)} \quad \frac{(\Delta, S) \Downarrow (\Delta',\ \text{no-return}) \quad (\Delta', S_s) \Downarrow (\Delta'', r)}{(\Delta, S; S_s) \Downarrow (\Delta'', r)}$$

Meaning: execute the left statement to completion (no return), then execute the right; the final result is the right-hand result.

Symbol	Meaning
Δ	Store / environment mapping variables to constants (memory)
⊢ (turnstile)	Judgment turnstile; relates a store and an expression/statement to an evaluation relation
⇓	Big-step evaluation relation: evaluates an expression/statement to a result in one big step

Symbol	Meaning
`(Δ, S) ⇓ (Δ', r)`	Configuration evaluation: statement `S` in store `Δ` evaluates to new store `Δ'` and result `r`
`no-return`	Marker meaning the statement completed normally with no return value
`r`	Result of evaluation — either a constant value or a `return` value
C	Constant literal (e.g., `0`, `true`, `false`)
X	Variable name

Notes:

- `no-return` indicates normal completion (no `return` was executed). When a rule yields a return value, `r` holds that value.
- Big-step rules summarize whole computations in a single inference; they are complementary to the small-step `→` relation.

Example of Big Step with SIMP

back-to-top

```
x = input;
x = x + 1;
return x;
```

Big Step Analysis

```
Δ₀ ⊢ input ⇓ v                       (assumption / external input)
(Δ₀, x = input) ⇓ (Δ₁, no-return)    (bAssign) — update store x ↦ v
Δ₁ ⊢ x ⇓ v                           (bVar)
Δ₁ ⊢ 1 ⇓ 1                           (bConst)
Δ₁ ⊢ x + 1 ⇓ v+1                     (bOp) — compute primitive result v+1
(Δ₁, x = x + 1) ⇓ (Δ₂, no-return)    (bAssign) — update store x ↦ v+1
(Δ₂, return x) ⇓ (Δ₂, v+1)           (bReturn) — return value v+1
(Δ₀, x = input;
x = x + 1; return x)
 ⇓
 (Δ₂, v+1)                            (bSeq) — full program result
```

Static Semantics (9A)

back-to-top

"What can we prove at compile time without running the program?"

A statically correct program must satisfy

1. All users of variables in it must be defined somewhere earlier
2. All the use of variables, the types must be matching with the expected type in the context

```
Statically correct   Statically incorrect
x = 0;               x = 0;
y = input;           y = input;
if y > x {           if y + x {  // type error
y = 0;               x = z;      // undefined variable z
return y;            }
}                    return x;
```

SIMP and Type environments

back-to-top

- Recall, SIMP has
  - Types: `T ::= int | bool`
  - Expressions: variables, constants, binary operators `(+ - * < ==>)` , parentheses
  - Statements: assignment, `return` , `nop` , `if E {...} else {...}` , `while E {...}`
- A **type environment** is:
  - `Γ ⊢ E : T` → *"Under environment Γ, expression E has type T."*
  - (Type env) `Γ : Var → Type` e.g. `Γ = { x : int, y : bool }`
- A **mapping (commonly written `Γ`)** from variable names to types, like a symbol table is:
  - `Γ = {(x,int) , (y,bool)}`

2 types of rules

Given a type environment `Γ` and a SIMP program `S` , we would like to check whether `S` is having a valid type assignment. Sometimes, we refer a program `S` that has valid type assignment is typeable, or well-typed.

Statement Typing

- $\Gamma \vdash \overline{S}$: checks whether the statement sequence $\overline{S}$ is well-typed under Γ (i.e. the program is typeable / well-typed).

Expression Typing

- $\Gamma \vdash E : T$: checks whether expression E has type T under the type environment Γ.

Type Checking Expressions

Expression typing rules

back-to-top

$$\textbf{(tVar)} \qquad \frac{(X, T) \in \Gamma}{\Gamma \vdash X : T}$$

Meaning: a variable has the type recorded for it in the type environment Γ.

$$\textbf{(tBool)} \qquad \frac{\{\text{C is an integer}\}}{\Gamma \vdash C : \text{bool}}$$

Meaning: integer literal `C` has type `int` .

$$\textbf{(tBool)} \qquad \frac{C \in \{\text{true},\ \text{false}\}}{\Gamma \vdash C : \text{bool}}$$

Meaning: boolean literals `true` and `false` have type `bool` .

$$(\textbf{tOp1}) \qquad \frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int} \quad \text{OP} \in \{+,\ -,\ *\}}{\Gamma \vdash E_1 \text{ OP } E_2 : \text{int}}$$

Meaning: arithmetic operators on two `int` operands produce an `int` .

$$(\textbf{tOp2}) \qquad \frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int} \quad \text{OP} \in \{==,\ <\}}{\Gamma \vdash E_1 \text{ OP } E_2 : \text{bool}}$$

Meaning: integer comparisons/equality produce a `bool` .

$$(\textbf{tOp3}) \qquad \frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : \text{bool} \quad \text{OP} \in \{==\}}{\Gamma \vdash E_1 \text{ OP } E_2 : \text{bool}}$$

Meaning: boolean equality produces `bool` . (I restricted `<` so it isn't applied to booleans.)

$$(\textbf{tParen}) \qquad \frac{\Gamma \vdash E : T}{\Gamma \vdash (E) : T}$$

Meaning: parentheses do not change the type.

Statement typing rules (checking statements)

back-to-top

$$(\textbf{tSeq}) \qquad \frac{\Gamma \vdash S \quad \Gamma \vdash \overline{S}}{\Gamma \vdash S\ \overline{S}}$$

Meaning: a sequence is well-typed when the first statement and the rest are well-typed.

$$(\textbf{tAssign}) \qquad \frac{\Gamma \vdash E : T \quad (X, T) \in \Gamma}{\Gamma \vdash X = E}$$

Meaning: assignment is well-typed when RHS has same type as the declared variable.

$$(\textbf{tReturn}) \qquad \frac{\Gamma \vdash X : T}{\Gamma \vdash \text{return } X}$$

Meaning: `return X` is well-typed when `X` has some type `T` under Γ.

$$(\textbf{tNop}) \qquad \frac{}{\Gamma \vdash \text{nop}}$$

Meaning: `nop` is always well-typed.

$$(\textbf{tIf}) \qquad \frac{\Gamma \vdash E : \text{bool} \quad \Gamma \vdash \overline{S_1} \quad \Gamma \vdash \overline{S_2}}{\Gamma \vdash \text{if } E\ \{\overline{S_1}\} \text{ else } \{\overline{S_2}\}}$$

Meaning: conditional is well-typed when guard is `bool` and both branches are well-typed.

$$(\textbf{tWhile}) \qquad \frac{\Gamma \vdash E : \text{bool} \quad \Gamma \vdash \overline{S}}{\Gamma \vdash \text{while } E\ \{\overline{S}\}}$$

Meaning: loop is well-typed if guard is `bool` and body is well-typed.

Example

back-to-top

The type environment (symbol table) maps `input` and `x` to type `int` "type check": verify every expression and statement in the program is well-typed under `Γ` (i.e. follow the typing rules you listed: `Γ ⊢ E : T` and `Γ ⊢ S`).

Recall

1. Γ (gamma) is the type environment (or context)
2. The typing rules (tVar, tAssign, tOp1, etc.) are separate schemas that consult Γ when checking an expression or statement.

*Refer to handwritten notes

Type Inference

back-to-top

Progress → a well typed SIMP Program must not be stuck (can run till return)

Preservation → as you run the program it will not change the typing of the program, program's well-typedness will not be altered by execution

Given a SIMP program $\overline{S}$ find the best Γ such that $\Gamma \vdash \overline{S}$

What defines "best"?

- Smallest possible Γ that makes $\overline{S}$ possible

In other words, this is the reverse of Type Checking

Meta Symbols for Type Inference

back-to-top

$$\begin{array}{rcl}
(\text{Extended types}) & \hat{\tau} & ::= \ \alpha \ | \ T \\
(\text{Constraints}) & \kappa & \subseteq \ \{\,(\hat{\tau}_1, \hat{\tau}_2)\,\} \\
(\text{Type substitution}) & \psi & ::= \ [\hat{\tau}/\alpha] \ | \ \psi_1 \circ \psi_2
\end{array}$$

Where:

- α denotes a type variable.
- $\hat{\tau}$ is an extended type (either a type variable α or a concrete type T, e.g. $\texttt{int}, \texttt{bool}$).
- A constraint set κ is a set of pairs of extended types required to be equal.
  Example: $\kappa = \{(\alpha, \beta), (\beta, \texttt{int})\}$ implies $\alpha = \texttt{int}$ after solving.
- ψ is a type substitution. Composition $\psi_1 \circ \psi_2$ means “apply ψ_2 then ψ_1”.

Type Substitution (definition / equations)

back-to-top

Type substitution $[\hat{\tau}/\alpha]$ replaces occurrences of the type variable α with $\hat{\tau}$.

$$\begin{aligned}
[\hat{\tau}/\alpha](\alpha) &= \hat{\tau} \\
[\hat{\tau}/\alpha](\beta) &= \beta \quad \text{if } \beta \neq \alpha \\
[\hat{\tau}/\alpha](T) &= T \quad \text{for base types } T \in \{\texttt{int}, \texttt{bool}, \dots\} \\
(\psi_1 \circ \psi_2)(\hat{\tau}) &= \psi_1\big(\psi_2(\hat{\tau})\big)
\end{aligned}$$

Note: apply substitutions elementwise to constraints; compose substitutions to combine solutions during unification.

Type Inference Judgments for SIMP (constraint-generation style)

back-to-top

- For statements: `S ⊢ κ` (statement `S` generates constraint set `κ`).
- For expressions: `E ⊢ \hat{\tau}, κ` (expression `E` has inferred type `\hat{\tau}` and generates constraints `κ`).

Expression rules (constraint generation)

- (iConst-int)

$$\frac{}{c \vdash \texttt{int},\ \emptyset} \qquad (c \text{ an integer literal})$$

- (iConst-bool)

$$\frac{}{b \vdash \texttt{bool},\ \emptyset} \qquad (b \in \{\texttt{true}, \texttt{false}\})$$

- (iVar)

$$\frac{}{X \vdash \alpha_X,\ \{(\alpha_X, \tau_X)\}}$$

Here α_X is a fresh type variable for this occurrence of `X`. If `X`'s declared type τ_X is known, this produces the constraint $\alpha_X = \tau_X$; otherwise τ_X may itself be unknown.

- (iOp) — arithmetic operators `+,-,*`

$$\frac{E_1 \vdash \hat{\tau}_1, \kappa_1 \quad E_2 \vdash \hat{\tau}_2, \kappa_2}{E_1 \text{ OP } E_2\ \vdash\ \hat{\tau},\ \kappa_1 \cup \kappa_2 \cup \{(\hat{\tau}_1, \texttt{int}), (\hat{\tau}_2, \texttt{int}), (\hat{\tau}, \texttt{int})\}}$$

- (iCmp) — comparisons `==, <` produce `bool`

$$\frac{E_1 \vdash \hat{\tau}_1, \kappa_1 \quad E_2 \vdash \hat{\tau}_2, \kappa_2}{E_1 \text{ OP } E_2\ \vdash\ \hat{\tau},\ \kappa_1 \cup \kappa_2 \cup \{(\hat{\tau}_1, \texttt{int}), (\hat{\tau}_2, \texttt{int}), (\hat{\tau}, \texttt{bool})\}}$$

- (iParen)

$$\frac{E \vdash \hat{\tau}, \kappa}{(E) \vdash \hat{\tau}, \kappa}$$

Statement rules (constraint generation)

- (iAssign)

$$\frac{E \vdash \hat{\tau}_E, \kappa_E}{(X = E) \vdash \kappa_E \cup \{(\hat{\tau}_E, \tau_X)\}}$$

Here τ_X is the declared type of `X` if known; otherwise treat τ_X as a fresh type variable representing `X`'s type.

- (iSeq)

$$\frac{S_1 \vdash \kappa_1 \quad S_2 \vdash \kappa_2}{(S_1; S_2) \vdash \kappa_1 \cup \kappa_2}$$

- (iIf)

$$\frac{E \vdash \hat{\tau}_E, \kappa_E \quad S_1 \vdash \kappa_1 \quad S_2 \vdash \kappa_2}{\text{if } E\{S_1\} \text{ else } \{S_2\} \vdash \kappa_E \cup \kappa_1 \cup \kappa_2 \cup \{(\hat{\tau}_E, \texttt{bool})\}}$$

- (iWhile)

$$\frac{E \vdash \hat{\tau}_E, \kappa_E \quad S \vdash \kappa_S}{\text{while } E\{S\} \vdash \kappa_E \cup \kappa_S \cup \{(\hat{\tau}_E, \texttt{bool})\}}$$

- (iNop)

$$\frac{}{\text{nop} \vdash \emptyset}$$

- (iReturn)

$$\frac{X \vdash \hat{\tau}_X, \kappa_X}{\text{return } X \vdash \kappa_X}$$

(Optionally record returned type in program-level constraints to ensure consistent return types.)

Type Inference Rules for SIMP

back-to-top
For statements S ⊨κ, S ⊨κ
For expression E ⊨ ˆT,κ

Notation: `E ⊢ τ̂; κ` means expression `E` has inferred (extended) type `τ̂` and generates constraint set `κ`.
`S ⊢ κ` means statement `S` generates constraint set `κ`. Type variables are `\alpha, \beta, ...`; extended types are `\hat{\tau}`; constraint sets are `\kappa`.

Expression rules

back-to-top

$$(\textbf{tiInt}) \quad \frac{C \text{ is an integer}}{C \vdash \texttt{int},\ \emptyset}$$

Meaning: integer literals are typed `int` and generate no constraints.

$$(\textbf{tiBool}) \quad \frac{C \in \{\texttt{true}, \texttt{false}\}}{C \vdash \texttt{bool},\ \emptyset}$$

Meaning: boolean literals are typed `bool` and generate no constraints.

$$(\textbf{tiVar}) \quad \frac{}{X \vdash \alpha_X,\ \emptyset}$$

Meaning: an occurrence of variable `X` is given a fresh type variable `\alpha_X` (to be constrained later).

$$(\textbf{tiOp1}) \quad \frac{E_1 \vdash \hat{\tau}_1, \kappa_1 \quad E_2 \vdash \hat{\tau}_2, \kappa_2 \quad \text{OP} \in \{+, -, *\}}{E_1 \text{ OP } E_2\ \vdash\ \texttt{int},\ \kappa_1 \cup \kappa_2 \cup \{(\hat{\tau}_1, \texttt{int}), (\hat{\tau}_2, \texttt{int})\}}$$

Meaning: arithmetic ops require both operands be `int`; collect operand constraints and return `int`.

$$(\textbf{tiOp2}) \quad \frac{E_1 \vdash \hat{\tau}_1, \kappa_1 \quad E_2 \vdash \hat{\tau}_2, \kappa_2 \quad \text{OP} \in \{<, ==\}}{E_1 \text{ OP } E_2\ \vdash\ \texttt{bool},\ \kappa_1 \cup \kappa_2 \cup \{(\hat{\tau}_1, \texttt{int}), (\hat{\tau}_2, \texttt{int})\}}$$

Meaning: comparisons require integer operands and yield `bool`; constrain operand types accordingly.

$$(\textbf{tiParen}) \quad \frac{E \vdash \hat{\tau}, \kappa}{(E) \vdash \hat{\tau}, \kappa}$$

Meaning: parentheses do not change the inferred type or constraints.

Statement rules

back-to-top

$$\textbf{(tiNop)} \qquad \frac{}{\text{nop} \vdash \emptyset}$$

Meaning: `nop` produces no constraints.

$$\textbf{(tiReturn)} \qquad \frac{}{\text{return } X \vdash \emptyset}$$

Meaning: a bare `return X` by itself produces no additional constraints here (you may include `X`'s expression constraints at program level).

$$\textbf{(tiSeq)} \qquad \frac{S_1 \vdash \kappa_1 \quad S_2 \vdash \kappa_2}{(S_1; S_2) \vdash \kappa_1 \cup \kappa_2}$$

Meaning: sequence accumulates constraints from both statements.

$$\textbf{(tiAssign)} \qquad \frac{E \vdash \hat{\tau}_E, \kappa_E}{(X = E) \vdash \kappa_E \cup \{(\alpha_X, \hat{\tau}_E)\}}$$

Meaning: assignment enforces the RHS type equals the variable's type (introduce a constraint linking `\alpha_X` and RHS).

$$\textbf{(tiIf)} \qquad \frac{E \vdash \hat{\tau}_E, \kappa_E \quad S_1 \vdash \kappa_1 \quad S_2 \vdash \kappa_2}{\text{if } E\{S_1\} \text{ else } \{S_2\} \vdash \kappa_E \cup \kappa_1 \cup \kappa_2 \cup \{(\hat{\tau}_E, \texttt{bool})\}}$$

Meaning: guard must be `bool`; collect constraints from guard and both branches.

$$\textbf{(tiWhile)} \qquad \frac{E \vdash \hat{\tau}_E, \kappa_E \quad S \vdash \kappa_S}{\text{while } E\{S\} \vdash \kappa_E \cup \kappa_S \cup \{(\hat{\tau}_E, \texttt{bool})\}}$$

Meaning: loop guard must be `bool`; collect constraints from guard and body.

Example

Refer to handwritten notes

Unification

back-to-top

Below are the basic definition and equations for the most-general unifier (mgu) used when solving type constraints.

$$\begin{aligned} \text{mgu}(\texttt{int}, \texttt{int}) &= [] \\ \text{mgu}(\texttt{bool}, \texttt{bool}) &= [] \\ \text{mgu}(\alpha, \hat{\tau}) &= [\hat{\tau}/\alpha] \quad (\text{if } \alpha \notin \text{fv}(\hat{\tau})) \\ \text{mgu}(\hat{\tau}, \alpha) &= [\hat{\tau}/\alpha] \quad (\text{symmetric}) \end{aligned}$$

Notes:

- `[]` denotes the identity substitution (no changes).
- `[\hat{\tau}/\alpha]` denotes the substitution that replaces the type variable `\alpha` with `\hat{\tau}`.
- The side-condition `\alpha \notin \mathrm{fv}(\hat{\tau})` is the occurs-check: it prevents creating infinite types (e.g. `\alpha = \alpha -> int`).

We overload `mgu` to operate on a set of type equations (a constraint set) by solving one equation at a time and composing substitutions:

$$\begin{aligned}\operatorname{mgu}(\emptyset) &= [\,] \\ \operatorname{mgu}(\{(\hat\tau_1,\hat\tau_2)\}\cup\kappa) &= \quad \text{let } \psi_1 = \operatorname{mgu}(\hat\tau_1,\hat\tau_2) \\ &\quad \kappa' = \psi_1(\kappa) \\ &\quad \psi_2 = \operatorname{mgu}(\kappa') \\ &\quad \text{in } \psi_2\circ\psi_1\end{aligned}$$

Explanation:

- Solve one pair `(\hat{\tau}_1,\hat{\tau}_2)` to get substitution `\psi_1` .
- Apply `\psi_1` to the remaining constraints `\kappa` to obtain `\kappa'` .
- Recursively compute `\psi_2 = \operatorname{mgu}(\kappa')` .
- Compose `\psi_2` after `\psi_1` to obtain the overall substitution that solves the whole constraint set.

Example (instantiation): given constraints `{(\alpha_x,\alpha_{input}), (\alpha_y,\mathtt{int}), (\alpha_y,\alpha_x)}` the unifier produced is `\{\alpha_x\mapsto\mathtt{int},\;\alpha_y\mapsto\mathtt{int},\;\alpha_{input}\mapsto\mathtt{int}\}` .

Type Inference Formal Results

back-to-top

- **Soundness** → Soundness means "the types we infer are correct": if the inference algorithm returns a type environment Γ for program S, then S really does typecheck under Γ (written Γ ⊢ S). In other words, inference never lies — it won't claim a program has a type that fails the ordinary type-checking rules.

- **Principality** → the inference algorithm yields the principal (most-general) type for an expression—any other valid type is its instance via substitution. Principal types maximize polymorphism, let other types be obtained by instantiation, and guarantee the inference result is unique up to renaming.

Limitations of the Type Inference Algorithm

back-to-top

1. Unspecified input type → e.g. x data type not declared
2. Uninitialised variable → e.g. x declared but never used
3. Control flow insensitivity → e.g. declare x as int then as bool

Static Semantics (Type Checking) (9B)

back-to-top

Simply Typed Lambda Calculus

Type checking = “Given a program + type hints, verify it’s well-typed.”

Type inference = “Given only the program, discover the best type hints that makes it well-typed.”

Type Checking Rules

$$(\textbf{lctInt}) \qquad \frac{c \text{ is an integer}}{\Gamma \vdash c : \mathtt{int}}$$

Meaning: If it looks like an integer number, it is type `int` .

$$\textbf{(lctBool)} \qquad \frac{c \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash c : \texttt{bool}}$$

Meaning: If it is `true` or `false` , it is type `bool` .

$$\textbf{(lctVar)} \qquad \frac{(x, T) \in \Gamma}{\Gamma \vdash x : T}$$

Meaning: To find a variable's type, look it up in our list of known variables (Γ).

$$\textbf{(lctLam)} \qquad \frac{\Gamma \oplus (x, T) \vdash t : T'}{\Gamma \vdash \lambda x : T.\, t : T \to T'}$$

Meaning: To check a function, assume the input `x` has type `T` , then check if the body `t` produces type `T'` .

$$\textbf{(lctApp)} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

Meaning: If you have a function expecting `T1` and you give it an input of type `T1` , it returns a `T2` .

$$\textbf{(lctLet)} \qquad \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \oplus (x, T_1) \vdash t_2 : T_2}{\Gamma \vdash \text{let } x : T_1 = t_1 \text{ in } t_2 : T_2}$$

Meaning: Check the first part `t1` . Then use its type to check the second part `t2` where `x` is now defined.

$$\textbf{(lctIf)} \qquad \frac{\Gamma \vdash t_1 : \texttt{bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

Meaning: The condition must be a boolean. The 'then' and 'else' branches must both return the exact same type.

$$\textbf{(lctOp1)} \qquad \frac{\Gamma \vdash t_1 : \texttt{int} \quad \Gamma \vdash t_2 : \texttt{int} \quad \text{op} \in \{+, -, *, /\}}{\Gamma \vdash t_1 \text{ op } t_2 : \texttt{int}}$$

Meaning: Math operations (+, -, *, /) need two integers and give back an integer.

$$\textbf{(lctOp2)} \qquad \frac{\Gamma \vdash t_1 : \texttt{int} \quad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 == t_2 : \texttt{bool}}$$

Meaning: Comparing two integers with `==` gives back a boolean (true/false).

$$\textbf{(lctOp3)} \qquad \frac{\Gamma \vdash t_1 : \texttt{bool} \quad \Gamma \vdash t_2 : \texttt{bool}}{\Gamma \vdash t_1 == t_2 : \texttt{bool}}$$

Meaning: Comparing two booleans with `==` gives back a boolean.

$$\textbf{(lctFix)} \qquad \frac{\Gamma \vdash t : (T_1 \to T_2) \to T_1 \to T_2}{\Gamma \vdash \text{fix } t : T_1 \to T_2}$$

Meaning: Used for recursion. If you have a function that takes a function and returns a function, `fix` finds the recursive solution.

Example for Type Checking

$$\Gamma \vdash t : T$$

Given a type environment `Γ` and lambda term `t` and a type `T` , check whether `t` can be given a type `T` under `Γ`

```
let f : int → int = (λx : Int.(x + 1)) inf 0
```

Goal: Check if `if true then 1 else 2` has type `int` .

Logic:

1. Check the condition `true`. It is a `bool`. (Uses **lctBool**)
2. Check the 'then' branch `1`. It is an `int`. (Uses **lctInt**)
3. Check the 'else' branch `2`. It is an `int`. (Uses **lctInt**)
4. Since both branches are `int`, the whole expression is `int`. (Uses **lctIf**)

Derivation:

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{true}:\texttt{bool}}(\textbf{lctBool}) \quad \dfrac{}{\emptyset \vdash 1:\texttt{int}}(\textbf{lctInt}) \quad \dfrac{}{\emptyset \vdash 2:\texttt{int}}(\textbf{lctInt})}{\emptyset \vdash \text{if } \texttt{true} \text{ then } 1 \text{ else } 2 : \texttt{int}}(\textbf{lctIf})$$

Simply Typed Calculus Type Checking Formal Properties

Progress
Let t be a simply typed lambda calculus term such that $fv(t) = \{\}$. Let T be a type such that $\{\} \vdash t : T$. Then t is either a value or there exists some t' such that $t \longrightarrow t'$.

Preservation
Let t and t' be simply typed lambda calculus terms such that $t \longrightarrow t'$. Let T be a type and Γ be a type environment such that $\Gamma \vdash t : T$. Then $\Gamma \vdash t' : T$.

Uniqueness
Let t be a simply typed lambda calculus term. Let Γ be a type environment such that for all $x \in fv(t)$, $x \in dom(\Gamma)$. Let T and T' be types such that $\Gamma \vdash t : T$ and $\Gamma \vdash t : T'$. Then T and T' must be the same.

Simply Typed Lambda Calculus Limitation

1. **Verbosity**
   - *Intuition:* You have to explicitly write down the type for every single variable and argument (e.g., `\lambda x: int`). It gets tedious quickly.
2. **Polymorphism is not supported**
   - *Intuition:* You cannot write a "generic" function. For example, the identity function `id(x) = x`. In STLC, you must define one for integers (`id_int`), one for booleans (`id_bool`), etc. You can't just say "this works for any type".
   - *Example below:* We want `f` to be generic (work for both `1` and `true`), but STLC forces us to pick one specific type for `x`, making this code invalid.

$$\begin{array}{l} \text{let } f = \lambda x : \alpha . x \\ \text{in let } g = \lambda x : \texttt{int}.\lambda y : \texttt{bool}.x \\ \text{in } (g\ (f\ 1)\ (f\ \texttt{true})) \end{array}$$

Where α denotes some generic type.

Hindley-Milner Type System (HM)

back-to-top

The **Hindley-Milner (HM)** type system is a classic type system that solves the limitations of the Simply Typed Lambda Calculus (STLC). It is the foundation for languages like Haskell, OCaml, and Standard ML.

Key Features

1. **Parametric Polymorphism (Generics):**
   - Unlike STLC, HM allows functions to be generic. You can write a function that works on *any* type `\alpha`.
   - *Example:* `id = \lambda x. x` has type `\forall \alpha. \alpha \to \alpha`. It works for `int`, `bool`, etc.
2. **Type Inference:**
   - You do not need to explicitly write down types. The compiler can deduce the most general type for every expression automatically.
   - This solves the **verbosity** problem of STLC.
3. **Principal Types:**
   - If a program is well-typed, HM guarantees that the inference algorithm will find the **most general** (principal) type. Any other valid type is just a specific instance of this principal type.
4. **Let-Polymorphism:**
   - The `let` construct is special in HM. It is the point where types are "generalized" (made generic).
   - `let f = ... in ...` allows `f` to be used with different types in the body (e.g., once as `int -> int` and once as `bool -> bool`).

How it works (Intuition)

HM uses **unification** (solving equations) to figure out types.

1. Assign a fresh type variable (like α, β) to every unknown.
2. Collect constraints based on how variables are used (e.g., if you see `x + 1`, then `x` must be `int`).
3. Solve these constraints to find the values of α, β.

Hindley Milner Syntax

We re-define the lambda calculus syntax for Hindley Milner Type System as follows:

$$\begin{array}{rrcl}
\text{(Lambda Terms)} & t & ::= & x \mid \lambda x.t \mid t\ t \mid \text{let}\ x = t\ \text{in}\ t \mid \text{if}\ t\ \text{then}\ t\ \text{else}\ t \mid t\ \text{op}\ t \mid c \mid \text{fix}\ t \\
\text{(Builtin Operators)} & op & ::= & + \mid - \mid * \mid / \mid == \\
\text{(Builtin Constants)} & c & ::= & 0 \mid 1 \mid \cdots \mid \text{true} \mid \text{false} \\
\text{(Types)} & T & ::= & \text{int} \mid \text{bool} \mid T \to T \mid \alpha \\
\text{(TypeScheme)} & \sigma & ::= & \forall \alpha.\sigma \mid T \\
\text{(Type Environments)} & \Gamma & \subseteq & (x \times \sigma) \\
\text{(Type Substitution)} & \Psi & ::= & [T/\alpha] \mid [] \mid \Psi \circ \Psi
\end{array}$$

In the above grammar rules, we remove the type annotations from the lambda abstraction and let binding.

Legend

Symbol	Name	Meaning
Γ	Type Environment	A map/list of variables and their known types (e.g., `{x: int, y: bool}`).
$\oplus$	Environment Update	$\Gamma \oplus (x, T)$ means "add variable x with type T to Γ, overwriting any previous x".
α, β	Type Variables	Placeholders for unknown types (like generics `T` in Java/C++).
T	Monotype	A specific, concrete type (e.g., `int`, `bool`, `int -> int`). No `\forall`.
σ	Type Scheme	A polymorphic type that may contain `\forall` (e.g., `\forall \alpha. \alpha \to \alpha`).
$\forall\alpha$	Quantifier	"For all types α". Indicates the type is generic.
$\sqsubseteq$	Instantiation	"Is more specific than". $\sigma_1 \sqsubseteq \sigma_2$ means σ_2 is a specific instance of generic σ_1.

Symbol	Name	Meaning
$ftv(T)$	Free Type Variables	The set of type variables in T that are not bound by a `\forall` .
Ψ	Substitution	A mapping that replaces type variables with types (e.g., replace α with `int`).

Hindley Milner Type System - Type Checking Mode

$$\textbf{(hmInt)} \quad \frac{c \text{ is an integer}}{\Gamma \vdash c : \text{int}}$$

Meaning: Integer literals are always type `int` .

$$\textbf{(hmBool)} \quad \frac{c \in \{\text{true}, \text{false}\}}{\Gamma \vdash c : \text{bool}}$$

Meaning: Boolean literals are always type `bool` .

$$\textbf{(hmVar)} \quad \frac{(x, \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

Meaning: Look up a variable in the environment. It might have a polymorphic type scheme σ.

$$\textbf{(hmLam)} \quad \frac{\Gamma \oplus (x, T) \vdash t : T'}{\Gamma \vdash \lambda x.t : T \to T'}$$

Meaning: Function definition. We guess a monomorphic type T for the argument x and check the body.

$$\textbf{(hmApp)} \quad \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

Meaning: Function application. Input type must match the function's expected argument type.

$$\textbf{(hmFix)} \quad \frac{(\text{fix}, \forall \alpha.(\alpha \to \alpha) \to \alpha) \in \Gamma}{\Gamma \vdash \text{fix} : \forall \alpha.(\alpha \to \alpha) \to \alpha}$$

Meaning: The fixed-point operator is generic. It works for any type α.

$$\textbf{(hmIf)} \quad \frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : \sigma \quad \Gamma \vdash t_3 : \sigma}{\Gamma \vdash \text{if } t_1 \{t_2\} \text{ else } \{t_3\} : \sigma}$$

Meaning: If-else. Condition is bool, branches must match.

$$\textbf{(hmOp1)} \quad \frac{\Gamma \vdash t_1 : \text{int} \quad \Gamma \vdash t_2 : \text{int} \quad op \in \{+, -, *, /\}}{\Gamma \vdash t_1 \text{ op } t_2 : \text{int}}$$

Meaning: Arithmetic operations take ints and return int.

$$\textbf{(hmOp2)} \quad \frac{\Gamma \vdash t_1 : \text{int} \quad \Gamma \vdash t_2 : \text{int}}{\Gamma \vdash t_1 == t_2 : \text{bool}}$$

Meaning: Integer comparison returns bool.

$$\textbf{(hmOp3)} \quad \frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : \text{bool}}{\Gamma \vdash t_1 == t_2 : \text{bool}}$$

Meaning: Boolean equality returns bool.

$$\textbf{(hmLet)} \quad \frac{\Gamma \vdash t_1 : \sigma_1 \quad \Gamma \oplus (x, \sigma_1) \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}$$

Meaning: **Let-Polymorphism**. We infer a (possibly generic) type scheme σ_1 for t_1, bind it to x, and then check t_2. This allows x to be used with different types inside t_2.

$$(\textbf{hmInst}) \qquad \frac{\Gamma \vdash t : \sigma_1 \quad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash t : \sigma_2}$$

Meaning: **Instantiation**. If variable x has a generic type (e.g., $\forall\alpha.\alpha \to \alpha$), we can use it as a specific type (e.g., $\text{int} \to \text{int}$).

$$(\textbf{hmGen}) \qquad \frac{\Gamma \vdash t : \sigma \quad \alpha \notin ftv(\Gamma)}{\Gamma \vdash t : \forall\alpha.\sigma}$$

Meaning: **Generalization**. If a type variable α is free in the type of t and not constrained by the environment Γ, we can make it generic (add $\forall\alpha$).

Note: $\sigma_1 \sqsubseteq \sigma_2$ iff $\sigma_1 = \forall\alpha.\sigma_1'$ and there exists a type substitution Ψ such that $\Psi(\sigma_1') = \sigma_2$.

Hindley Milner Type System Formal Properties

Progress

Let t be a lambda calculus term such that $fv(t) = \{\}$. Let σ be a type scheme such that $\Gamma_{init} \vdash t : \sigma$. Then t is either a value or there exists some t' such that $t \longrightarrow t'$.

Preservation

Let t and t' be lambda calculus terms such that $t \longrightarrow t'$. Let σ be a type scheme and Γ be a type environment such that $\Gamma \vdash t : \sigma$. Then $\Gamma \vdash t' : \sigma$.

Uniqueness

The following property states that if a lambda term is typable, its type scheme must be unique modulo type variable renaming.
Let t be a lambda calculus term. Let Γ be a type environment such that for all $x \in fv(t)$, $x \in dom(\Gamma)$. Let σ and σ' be type schemes such that $\Gamma \vdash t : \sigma$ and $\Gamma \vdash t : \sigma'$. Then σ and σ' must be the same modulo type variable renaming.

Hindley Milner Inference Mode

In theory, we could use the Hindler Milner rules for type inference too, except:

- `fix` type must be given in the initial type environment (not a big deal!)
- The `(hmGen)` and `(hmInst)` make the system non-syntax directed, i.e. huge search space.

HMTS Example

Infer the type of the expression using the Hindley-Milner rules

```
Let Γ = {}.
// For recursion,
// Let Γ = { (fix, ∀α. (α → α) → α) }

let f = λx.x
  in let g = λx.λy.x
    in (g (f 1) (f true))
```

Walkthrough

Step 0: Initial State

- `Γ = {}`
- This means our **Type Environment is empty**. We have no variables or functions defined yet. We are starting from scratch.

Step 1: Analyze `f = λx.x`

- We need to find the type of `λx.x` .
- We assign a fresh **type variable** (placeholder) α to the argument x. So $x : \alpha$.
- The body is just x, so the body has type α.
- Therefore, `λx.x` has type $\alpha \to \alpha$.
- **Generalization (hmGen):**
  - *Why here?* Because we are defining `f` inside a **`let`** binding. In HM, `let` is the only time we are allowed to make types generic.
  - *Condition:* Is α free? Yes, Γ is empty, so α is not "locked" to any existing variable.
  - *Result:* We generalize $\alpha \to \alpha$ to the **Type Scheme** $\sigma_f = \forall \alpha . \alpha \to \alpha$.
- **Update Environment:** We add f to Γ.
  - New $\Gamma = \{f : \forall \alpha . \alpha \to \alpha\}$.

Step 2: Analyze `g = λx.λy.x`

- We assign fresh type variables to arguments: $x : \beta$ and $y : \gamma$.
- The body is x, which has type β.
- The function takes x (type β), then y (type γ), and returns x (type β).
- So `g` has type $\beta \to \gamma \to \beta$.
- **Generalization:** β and γ are generic, so we generalize.
  - $\sigma_g = \forall \beta, \gamma . \beta \to \gamma \to \beta$.
- **Update Environment:** Add g to Γ.
  - New $\Gamma = \{f : \forall \alpha . \alpha \to \alpha, \quad g : \forall \beta, \gamma . \beta \to \gamma \to \beta\}$.

Step 3: Analyze the Body `(g (f 1) (f true))`

This is where the magic happens. We use `f` twice with different types!

1. **Analyze `f 1` :**
   - Look up `f` in Γ: $\forall \alpha . \alpha \to \alpha$.
   - **Instantiation (hmInst):** We need to use `f` with an integer. So we replace generic α with `int` .
   - Type of `f` here becomes: `int -> int` .
   - Apply to `1` (int): Result type is **`int`** .
2. **Analyze `f true` :**
   - Look up `f` in Γ: $\forall \alpha . \alpha \to \alpha$.
   - **Instantiation (hmInst):** We need to use `f` with a boolean. So we replace generic α with `bool` .
   - Type of `f` here becomes: `bool -> bool` .
   - Apply to `true` (bool): Result type is **`bool`** .
3. **Analyze `g` :**
   - Look up `g` in Γ: $\forall \beta, \gamma . \beta \to \gamma \to \beta$.
   - **Instantiation (hmInst):** We are calling `g` with an `int` (from step 1) and a `bool` (from step 2).
   - We replace β with `int` and γ with `bool` .
   - Type of `g` here becomes: `int -> bool -> int` .
4. **Final Application:**
   - `g` takes an `int` (which is `f 1`) -> Returns a function `bool -> int` .
   - That function takes a `bool` (which is `f true`) -> Returns an `int` .
   - **Final Result:** The whole expression has type **`int`** .

Algorithm W

back-to-top

Goal: Given a type environment Γ and an expression t, find the **Principal Type** of t and a **Substitution** Ψ needed to make it valid.

Signature: The rules are in the form of $\Gamma, t \vDash T, \Psi$

- **Input:**
  - Γ: (starting) type environment.
  - t: the lambda expression.
- **Output:**
  - T: the inferred type.
  - Ψ: a type substitution (the solution to the constraints).

Algorithm W Rules

$$(\mathbf{wInt}) \quad \frac{c \text{ is an integer}}{\Gamma, c \vDash \mathtt{int}, []}$$

Meaning: Integer literals are always `int`. No substitution needed.

$$(\mathbf{wBool}) \quad \frac{c \in \{\mathtt{true}, \mathtt{false}\}}{\Gamma, c \vDash \mathtt{bool}, []}$$

Meaning: Boolean literals are always `bool`. No substitution needed.

$$(\mathbf{wVar}) \quad \frac{(x, \sigma) \in \Gamma \quad inst(\sigma) = T}{\Gamma, x \vDash T, []}$$

Meaning: Look up variable `x`. If it has a generic scheme σ, instantiate it to a fresh type T.

$$(\mathbf{wFix}) \quad \frac{(\text{fix}, \forall\alpha.(\alpha \to \alpha) \to \alpha) \in \Gamma \quad T = inst(\forall\alpha.(\alpha \to \alpha) \to \alpha)}{\Gamma, \text{fix} \vDash T, []}$$

Meaning: `fix` is a special variable with a known generic type. Instantiate it.

$$(\mathbf{wLam}) \quad \frac{\alpha_1 = newvar \quad \Gamma \oplus (x, \alpha_1), t \vDash T, \Psi}{\Gamma, \lambda x.t \vDash \Psi(\alpha_1 \to T), \Psi}$$

Meaning: To infer `\x.t`:

1. Create a fresh type var α_1 for `x`.
2. Infer the body `t` using that assumption.
3. Result is $\alpha_1 \to T$ (applying any learned substitutions Ψ to α_1).

$$(\mathbf{wLet}) \quad \frac{\Gamma, t_1 \vDash T_1, \Psi_1 \quad \Psi_1(\Gamma) \oplus (x, gen(\Psi_1(\Gamma), T_1)), t_2 \vDash T_2, \Psi_2}{\Gamma, \text{let } x = t_1 \text{ in } t_2 \vDash T_2, \Psi_2 \circ \Psi_1}$$

Meaning:

1. Infer type of t_1 (getting T_1, Ψ_1).
2. Generalize T_1 (make it generic) relative to the environment.
3. Infer t_2 with x bound to that generic scheme.
4. Combine substitutions.

$$(\mathbf{wApp}) \quad \frac{\Gamma, t_1 \vDash T_1, \Psi_1 \quad \Psi_1(\Gamma), t_2 \vDash T_2, \Psi_2 \quad \alpha_3 = newvar \quad \Psi_3 = mgu(\Psi_2(T_1), T_2 \to \alpha_3)}{\Gamma, (t_1\ t_2) \vDash \Psi_3(\alpha_3), \Psi_3 \circ \Psi_2 \circ \Psi_1}$$

Meaning:

1. Infer function t_1.
2. Infer argument t_2 (using updated env from step 1).
3. Create fresh var α_3 for the result.
4. Unify: Is T_1 equivalent to $T_2 \to \alpha_3$? (Does function accept argument?).
5. Result is α_3.

$$\textbf{(wOp1)} \qquad \frac{op \in \{+,-,*,/\} \quad \Gamma, t_1 \vDash T_1, \Psi_1 \quad \Psi_1(\Gamma), t_2 \vDash T_2, \Psi_2 \quad mgu(\Psi_2(T_1), T_2, \texttt{int}) = \Psi_3}{\Gamma, t_1 \text{ op } t_2 \vDash \texttt{int}, \Psi_3 \circ \Psi_2 \circ \Psi_1}$$

Meaning: Arithmetic operations. Infer both operands, then unify both of them with `int`.

$$\textbf{(wOp2)} \qquad \frac{\Gamma, t_1 \vDash T_1, \Psi_1 \quad \Psi_1(\Gamma), t_2 \vDash T_2, \Psi_2 \quad mgu(\Psi_2(T_1), T_2) = \Psi_3}{\Gamma, t_1 == t_2 \vDash \texttt{bool}, \Psi_3 \circ \Psi_2 \circ \Psi_1}$$

Meaning: Equality check. Infer both operands, then unify them (they must be the same type). Result is `bool`.

$$\textbf{(wIf)} \qquad \frac{\Gamma, t_1 \vDash T_1, \Psi_1 \quad \Psi_1' = mgu(\texttt{bool}, T_1) \circ \Psi_1 \quad \Psi_1'(\Gamma), t_2 \vDash T_2, \Psi_2 \quad \Psi_1'(\Gamma), t_3 \vDash T_3, \Psi_3 \quad \Psi_4 = mgu(\Psi_3(T_2), \Psi_2(T_3))}{\Gamma, \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \vDash \Psi_4(\Psi_3(T_2)), \Psi_4 \circ \Psi_3 \circ \Psi_2 \circ \Psi_1'}$$

Meaning:

1. Infer condition t_1. Unify with `bool`.
2. Infer `then` branch t_2.
3. Infer `else` branch t_3.
4. Unify types of t_2 and t_3 (branches must match).

Auxiliary Functions

The rules above rely on several helper functions:

1. **`newvar` (Fresh Variable):**
   - Returns a brand new type variable that has never been used before (e.g., α_{101}).
   - Crucial to avoid accidental name collisions between different parts of the code.
2. **`inst(σ)` (Instantiation):**
   - Takes a **Type Scheme** $\sigma = \forall \alpha_1 \dots \alpha_n . \tau$.
   - Replaces each bound variable α_i with a **fresh** type variable β_i (using `newvar`).
   - Returns the resulting Monotype τ'.
   - *Example:* `inst(∀α. α -> α)` might return `β -> β`.
3. **`gen(Γ, τ)` (Generalization):**
   - Takes a type environment Γ and a type τ.
   - Finds all type variables in τ that are **not** mentioned in Γ (i.e., $\alpha \in ftv(\tau) \setminus ftv(\Gamma)$).
   - Adds $\forall$ quantifiers for those variables.
   - *Example:* If $\Gamma = \{x : \text{int}\}$ and $\tau = \beta \to \beta$, then `gen` returns $\forall \beta . \beta \to \beta$.
4. **`mgu(τ1, τ2)` (Unification):**
   - Finds the "Most General Unifier" substitution that makes τ_1 and τ_2 identical.
   - If they cannot be unified (e.g., `int` vs `bool`), it fails.
5. **`ftv(τ)` or `ftv(Γ)` (Free Type Variables):**
   - Returns the set of type variables that are not bound by a $\forall$.

Algorithm W - Examples & Walkthrough

Q: What are we doing here? Is it type checking?

A: Not exactly. It is **Type Inference**.

- **Type Checking (⊢):** "Here is the code and the types. Are they correct?" (Yes/No).
- **Type Inference (⊨):** "Here is the code. I don't know the types. Can you calculate them for me?" (Returns the Type or Error).

Algorithm W is the **implementation** of the logic we discussed earlier. It is a recursive function that builds the type tree step-by-step.