IPFLang Type System Manual for Beginners

A Complete Guide to Reading and Understanding Formal Typing Rules

This manual is designed for readers with no prior exposure to formal type systems or mathematical logic notation. We will build your understanding from the ground up, starting with the basic symbols and gradually working through every typing rule in IPFLang.

Part 1: The Alphabet - Learning to Read the Symbols

Before we can read typing rules, we need to understand the symbols. Think of this like learning a new alphabet.

1.1 Greek Letters and Their Meanings

Symbol	Name	What It Represents
Γ	Gamma	The "typing environment" - a table that remembers what type each variable has
τ	Tau	A type (like `Num`, `Bool`, or `Amt[EUR]`)
σ	Sigma	A "type scheme" - a type that might have variables in it
α	Alpha	A type variable - a placeholder for an unknown type
φ	Phi	A condition or Boolean expression
ε	Epsilon	"Nothing" or "empty"

1.2 Mathematical Symbols

Symbol	Name	Meaning
⊢	Turnstile	"proves" or "entails" - means "we can determine that"
:	Colon	"has type" - connects an expression to its type
∈	Element of	"is a member of" or "belongs to"
∉	Not element of	"is not a member of"
∀	For all	"for every possible value of"
∃	Exists	"there is at least one"
∪	Union	Combines two sets together
∧	And	Logical AND (both must be true)
∨	Or	Logical OR (at least one must be true)
¬	Not	Logical negation

Symbol	Name	Meaning
→	Arrow	"maps to" or "becomes"
↦	Maps to	Used for substitution (x ↦ τ means "x is assigned type τ")
⟨ ⟩	Angle brackets	Used for currency annotation, like `100<EUR>` written as `100⟨EUR⟩`

1.3 Set Notation

Notation	Meaning
{a, b, c}	A set containing elements a, b, and c
{x : condition}	The set of all x where the condition is true
A × B	Cartesian product - all pairs (a, b) where a ∈ A and b ∈ B

1.4 IPFLang-Specific Types

Type	What It Represents
`Num`	A dimensionless number (like 42, 3.14, or 0)
`Bool`	A Boolean value (`TRUE` or `FALSE`)
`Sym`	A symbolic identifier (like `SmallEntity` from a LIST)
`Date`	A date value
`SymList`	A list of symbols (from a MULTILIST)
`Amt[c]`	A monetary amount in currency `c` (like `Amt[EUR]` or `Amt[USD]`)

Part 2: The Grammar - How to Read Inference Rules

2.1 The Basic Structure

Every typing rule follows this structure:

```
            premise₁   premise₂   ...   premiseₙ
[RULE-NAME] ──────────────────────────────────────
                        conclusion
```

How to read this:

> "IF all the premises above the line are true, THEN the conclusion below the line is true."

The horizontal line acts like a logical "therefore" - everything above must hold for the thing below to be valid.

2.2 A Simple Example

Let's look at a made-up simple rule:

```
         x is a dog
[DOG-ANIMAL] ──────────────
         x is an animal
```

This reads as: "If x is a dog, then x is an animal."

2.3 The Typing Judgment: Γ ⊢ e : τ

The most important notation in type systems is the **typing judgment**:

```
Γ ⊢ e : τ
```

Breaking this down piece by piece:

Part	Meaning
Γ	The typing environment (what we already know about variable types)
⊢	"proves" or "allows us to determine"
e	An expression (a piece of code)
:	"has type"
τ	The type we're claiming the expression has

In plain English: "Given what we know from the environment Γ, we can determine that expression e has type τ."

2.4 The Typing Environment (Γ)

Think of Γ as a lookup table:

```
Γ = {
    ClaimCount : Num,
    EntityType : Sym,
    FilingFee : Amt[EUR],
    IsUrgent : Bool
}
```

When we write `Γ(x) = τ`, we mean "looking up x in the table gives us type τ."

For example: `Γ(ClaimCount) = Num` means "ClaimCount has type Num according to our environment."

2.5 Extending the Environment

Sometimes we need to add new information to Γ. We write this as:

```
Γ[x ↦ τ]
```

This means "Γ with the additional knowledge that x has type τ."

If Γ = {a : Num}, then Γ[b ↦ Bool] = {a : Num, b : Bool}.

2.6 Rules with No Premises (Axioms)

Some rules have nothing above the line:

```
[RULE-NAME] ─────────────
            conclusion
```

These are **axioms** - things that are always true without needing any prerequisites.

Part 3: The Type Language - What Types Exist

Before diving into the rules, let's understand what types IPFLang has:

3.1 Base Types

```
τ ::= Num           -- dimensionless numbers (counts, ratios)
    | Bool          -- Boolean values (TRUE, FALSE)
    | Sym           -- symbolic identifiers (LIST choices)
    | Date          -- date values
    | SymList       -- symbol lists (MULTILIST selections)
    | Amt[c]        -- monetary amounts in currency c
    | α             -- type variables (placeholders)
```

The `::=` notation means "is defined as" and the `|` means "or". So this reads: "A type τ is either Num, or Bool, or Sym, or Date, or SymList, or Amt[c], or α."

3.2 Currency-Parameterized Types

`Amt[c]` is special - the `c` is a parameter that specifies which currency:

- `Amt[EUR]` = amount in Euros
- `Amt[USD]` = amount in US Dollars
- `Amt[JPY]` = amount in Japanese Yen

The `c` must be one of the 161 ISO-4217 currency codes.

3.3 Polymorphic Types (Type Schemes)

```
σ ::= τ           -- a plain type
    | ∀α. σ       -- "for all α, σ"
```

The `∀α. Amt[α]` notation means "for any currency α, an amount in that currency." This is a **generic** or **polymorphic** type that works with any currency.

Part 4: The Typing Rules Explained

Now we'll go through every typing rule in IPFLang. For each rule, I'll show:

1. The formal rule
2. A plain English explanation
3. Why the rule exists
4. Examples

Category 1: Literals and Variables

These rules handle the simplest expressions - constants and variable references.

Rule T-NUM (Numeric Literals)

```
[T-NUM] ──────────────
        Γ ⊢ n : Num
```

What the symbols mean:

- `n` is a numeric literal (a number you write directly in code)
- `Num` is the type for dimensionless numbers
- No premises above the line (this is an axiom)

Plain English: "Any numeric literal like 42, 3.14, or 0 has type Num."

Why this rule exists: We need to assign types to the most basic building blocks. Numbers without currency annotations are dimensionless - they represent counts, ratios, or multipliers, not money.

Examples:

- `42` has type `Num`
- `3.14159` has type `Num`
- `0` has type `Num`

Rule T-VAR (Variable Lookup)

```
        Γ(x) = τ
[T-VAR] ──────────
        Γ ⊢ x : τ
```

What the symbols mean:

- `Γ(x) = τ` means "looking up x in the environment gives type τ"
- `Γ ⊢ x : τ` means "therefore, x has type τ"

Plain English: "If the environment says variable x has type τ, then x has type τ."

Why this rule exists: When you reference a variable, its type comes from how it was declared. The environment Γ records all declarations.

Examples:

Given environment:

```
Γ = { ClaimCount : Num, EntityType : Sym, BaseFee : Amt[EUR] }
```

- `ClaimCount` has type `Num` (because Γ(ClaimCount) = Num)
- `EntityType` has type `Sym` (because Γ(EntityType) = Sym)
- `BaseFee` has type `Amt[EUR]` (because Γ(BaseFee) = Amt[EUR])

Rule T-CURR (Currency Literals)

```
           c ∈ ISO-4217
[T-CURR] ──────────────────
         Γ ⊢ n⟨c⟩ : Amt[c]
```

What the symbols mean:

- `c ∈ ISO-4217` means "c is a valid ISO-4217 currency code"
- `n⟨c⟩` represents a currency literal like `100<EUR>` (written `100⟨EUR⟩` formally)
- `Amt[c]` is an amount denominated in currency c

Plain English: "If c is a valid currency code, then a number annotated with that currency (like 100) has type Amt[c]."

Why this rule exists: This is where currency information enters the system. By explicitly annotating numbers with currencies, we can track what currency each value is in throughout the program.

Examples:

- `100<EUR>` has type `Amt[EUR]` (100 Euros)
- `50.50<USD>` has type `Amt[USD]` (50.50 US Dollars)

- `1000<JPY>` has type `Amt[JPY]` (1000 Japanese Yen)

Invalid examples:

- `100<XYZ>` would be rejected because `XYZ` is not a valid ISO-4217 code

---

Category 2: Arithmetic Operations

These rules govern how mathematical operations work with numbers and currency amounts.

Rule T-ADD-NUM (Adding Numbers)

$$[\text{T-ADD-NUM}]\ \frac{\Gamma \vdash e_1 : \text{Num} \qquad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash e_1 + e_2 : \text{Num}}$$

What the symbols mean:

- $\Gamma \vdash e_1 : \text{Num}$ means "e_1 has type Num"
- $\Gamma \vdash e_2 : \text{Num}$ means "e_2 has type Num"
- $e_1 + e_2$ is the addition of these expressions
- The conclusion says the sum also has type `Num`

Plain English: "If both operands are dimensionless numbers, their sum is also a dimensionless number."

Why this rule exists: Adding two counts gives you another count. Adding two ratios gives another ratio. There's no currency involved.

Examples:

- `ClaimCount + 10` where ClaimCount : Num → result has type `Num`
- `5 + 3` → type `Num`

Rule T-ADD-AMT (Adding Currency Amounts)

$$[\text{T-ADD-AMT}]\ \frac{\Gamma \vdash e_1 : \text{Amt}[c] \qquad \Gamma \vdash e_2 : \text{Amt}[c]}{\Gamma \vdash e_1 + e_2 : \text{Amt}[c]}$$

What the symbols mean:

- Both e_1 and e_2 must have the **same** currency type `Amt[c]`
- The result also has type `Amt[c]`

Plain English: "You can only add amounts that are in the SAME currency. The result is an amount in that same currency."

Why this rule exists: This is the **core safety guarantee** of IPFLang. Adding 100 EUR + 50 EUR makes sense (you get 150 EUR). But adding 100 EUR + 50 USD is meaningless without knowing the exchange rate. The type system **prevents this at compile time**.

Valid examples:

- `100<EUR> + 50<EUR>` → type `Amt[EUR]` ✓
- `BaseFee + ExtraFee` where both have type `Amt[USD]` → type `Amt[USD]` ✓

Invalid examples (rejected by type checker):

- `100<EUR> + 50<USD>` → **TYPE ERROR**: can't add different currencies
- `100<EUR> + 50` → **TYPE ERROR**: can't add currency to dimensionless number

---

Rule T-SUB-NUM (Subtracting Numbers)

$$[\text{T-SUB-NUM}]\ \frac{\Gamma \vdash e_1 : \text{Num} \qquad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash e_1 - e_2 : \text{Num}}$$

Plain English: "Subtracting two dimensionless numbers gives a dimensionless number."

Examples:

- `ClaimCount - 15` → type `Num`

Rule T-SUB-AMT (Subtracting Currency Amounts)

$$[\text{T-SUB-AMT}]\ \frac{\Gamma \vdash e_1 : \text{Amt}[c] \qquad \Gamma \vdash e_2 : \text{Amt}[c]}{\Gamma \vdash e_1 - e_2 : \text{Amt}[c]}$$

Plain English: "You can only subtract amounts in the SAME currency. The result is in that currency."

Why this rule exists: Same reasoning as addition. You can compute a difference between two EUR amounts, but EUR minus USD is meaningless.

Examples:

- `500<EUR> - 100<EUR>` → type `Amt[EUR]` (represents 400 EUR)
- `TotalFee - DiscountAmount` where both are `Amt[USD]` → type `Amt[USD]`

Rule T-MUL-NUM (Multiplying Numbers)

$$\text{[T-MUL-NUM]}\ \frac{\Gamma \vdash e_1 : \text{Num} \quad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash e_1 \times e_2 : \text{Num}}$$

Plain English: "Multiplying two dimensionless numbers gives a dimensionless number."

Examples:

- `ClaimCount * PageCount` → type `Num`
- `2 * 3` → type `Num`

Rule T-MUL-SCALAR-R (Multiplying Amount by Number - Right)

$$\text{[T-MUL-SCALAR-R]}\ \frac{\Gamma \vdash e_1 : \text{Amt}[c] \quad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash e_1 \times e_2 : \text{Amt}[c]}$$

What the symbols mean:

- e_1 is a currency amount
- e_2 is a dimensionless number (a "scalar")
- The result preserves the currency

Plain English: "Multiplying a currency amount by a number gives a currency amount in the same currency."

Why this rule exists: This is how you compute things like "fee per claim × number of claims". The count has no currency, but the fee does. Multiplying them gives a total fee in the same currency.

Examples:

- `100<EUR> * 2` → type `Amt[EUR]` (represents 200 EUR)
- `ClaimFee * ExcessClaimCount` where ClaimFee : Amt[EUR], ExcessClaimCount : Num → type `Amt[EUR]`

Rule T-MUL-SCALAR-L (Multiplying Number by Amount - Left)

$$\text{[T-MUL-SCALAR-L]}\ \frac{\Gamma \vdash e_1 : \text{Num} \quad \Gamma \vdash e_2 : \text{Amt}[c]}{\Gamma \vdash e_1 \times e_2 : \text{Amt}[c]}$$

Plain English: "Multiplying a number by a currency amount gives a currency amount."

Why we need both rules: Multiplication is commutative (a × b = b × a), so we need rules for both orders. Both `100<EUR> * 2` and `2 * 100<EUR>` should work.

Examples:

- `2 * 100<EUR>` → type `Amt[EUR]`
- `ExcessClaimCount * ClaimFee` → type `Amt[EUR]`

Rule T-DIV-NUM (Dividing Numbers)

$$[\text{T-DIV-NUM}]\ \frac{\Gamma \vdash e_1 : \text{Num} \qquad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash e_1 \,/\, e_2 : \text{Num}}$$

Plain English: "Dividing two dimensionless numbers gives a dimensionless number."

Examples:

- `10 / 2` → type `Num`
- `ClaimCount / 2` → type `Num`

Rule T-DIV-AMT-SCALAR (Dividing Amount by Number)

$$[\text{T-DIV-AMT-SCALAR}]\ \frac{\Gamma \vdash e_1 : \text{Amt}[c] \qquad \Gamma \vdash e_2 : \text{Num}}{\Gamma \vdash e_1 \,/\, e_2 : \text{Amt}[c]}$$

Plain English: "Dividing a currency amount by a number gives a currency amount."

Why this rule exists: This handles cases like splitting a fee among multiple applicants, or computing a per-unit cost.

Examples:

- `100<EUR> / 2` → type `Amt[EUR]` (50 EUR)
- `TotalFee / NumberOfApplicants` → type `Amt[EUR]`

What's NOT allowed:

- `10 / 100<EUR>` → No rule for this! Dividing a number by a currency would give "inverse currency" which is meaningless.

Category 3: Currency Conversion

Rule T-CONVERT

$$\text{[T-CONVERT]}\ \frac{\Gamma \vdash e : \text{Amt}[c] \quad c_1 = c \quad c_1 \in \text{ISO-4217} \quad c_2 \in \text{ISO-4217}}{\Gamma \vdash \text{CONVERT}(e, c_1, c_2) : \text{Amt}[c_2]}$$

What the symbols mean:

- $\Gamma \vdash e : \text{Amt}[c]$ - expression e has some currency type Amt[c]
- $c_1 = c$ - the declared source currency must match the actual currency
- $c_1 \in \text{ISO-4217}$ - source currency must be valid
- $c_2 \in \text{ISO-4217}$ - target currency must be valid
- Result type is $\text{Amt}[c_2]$ - an amount in the target currency

Plain English: "CONVERT changes a currency amount from one currency to another. The declared source currency must match the expression's actual currency."

Why this rule exists: This is the ONLY way to change currencies in IPFLang. Every currency conversion must be explicit and visible in the code. This ensures:

1. No accidental conversions happen silently
2. Auditors can see exactly where conversions occur
3. The programmer acknowledges the actual currency being converted

Why require $c_1 = c$? This prevents bugs like:

```
CONVERT(100<EUR>, USD, GBP)  -- ERROR: expression is EUR, not USD!
```

The programmer must correctly identify what they're converting FROM.

Examples:

- $\text{CONVERT}(100\langle\text{EUR}\rangle, \text{EUR}, \text{USD})$ → type $\text{Amt}[\text{USD}]$ ✓
- $\text{CONVERT}(\text{PriorFee}, \text{EUR}, \text{USD})$ where PriorFee : Amt[EUR] → type $\text{Amt}[\text{USD}]$ ✓

Invalid examples:

- $\text{CONVERT}(100\langle\text{EUR}\rangle, \text{USD}, \text{GBP})$ → ERROR: declared source (USD) ≠ actual (EUR)

Category 4: Comparisons and Conditions

Rule T-COMP-EQ (Equality Comparison)

$$\text{[T-COMP-EQ]}\ \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\text{Num}, \text{Bool}, \text{Sym}, \text{Date}\} \cup \{\text{Amt}[c] : c \in \text{ISO-4217}\} \quad \oplus \in \{\text{EQ}, \text{NEQ}\}}{\Gamma \vdash e_1 \oplus e_2 : \text{Bool}}$$

What the symbols mean:

- `τ ∈ {Num, Bool, Sym, Date} ∪ {Amt[c] : ...}` - τ must be a type that supports equality
- `⊕ ∈ {EQ, NEQ}` - the operator is either EQ (equals) or NEQ (not equals)
- Result type is `Bool`

Plain English: "You can compare two values of the SAME type for equality or inequality, and the result is a Boolean."

What types support equality:

- `Num` - you can check if two numbers are equal
- `Bool` - you can check if two Booleans are equal
- `Sym` - you can check if two symbols are equal (like `EntityType EQ SmallEntity`)
- `Date` - you can check if two dates are the same
- `Amt[c]` - you can check if two amounts (in the same currency) are equal

Examples:

- `ClaimCount EQ 10` → type `Bool`
- `EntityType EQ SmallEntity` → type `Bool`
- `BaseFee EQ 100<EUR>` where BaseFee : Amt[EUR] → type `Bool`

Invalid examples:

- `100 EQ TRUE` → ERROR: can't compare Num with Bool
- `100<EUR> EQ 100<USD>` → ERROR: different currency types

Rule T-COMP-ORD (Ordering Comparison)

```
        Γ ⊢ e₁ : τ   Γ ⊢ e₂ : τ   τ ∈ {Num, Date} ∪ {Amt[c] : c ∈ ISO-4217}   ⊕
∈ {GT, LT, GTE, LTE}
[T-COMP-ORD]
_____________________________________________________________________________
__________
                                          Γ ⊢ e₁ ⊕ e₂ : Bool
```

Plain English: "Ordering comparisons (greater than, less than, etc.) work only on types that have a natural order."

What types support ordering:

- `Num` - numbers can be compared (5 > 3)
- `Date` - dates can be compared (is filing date after priority date?)
- `Amt[c]` - amounts in the same currency can be compared (is fee > 100 EUR?)

What types DON'T support ordering:

- `Bool` - TRUE > FALSE makes no sense
- `Sym` - SmallEntity > LargeEntity makes no sense (no inherent order)

Examples:

- `ClaimCount GT 15` → type `Bool`
- `TotalFee GTE 1000<EUR>` where TotalFee : Amt[EUR] → type `Bool`
- `FilingDate GT PriorityDate` → type `Bool`

Invalid examples:

- `EntityType GT SmallEntity` → ERROR: Sym doesn't support ordering
- `TRUE GT FALSE` → ERROR: Bool doesn't support ordering

Rule T-AND (Logical AND)

$$[\text{T-AND}]\ \frac{\Gamma \vdash \phi_1 : \text{Bool} \quad \Gamma \vdash \phi_2 : \text{Bool}}{\Gamma \vdash \phi_1 \text{ AND } \phi_2 : \text{Bool}}$$

Plain English: "ANDing two Boolean expressions gives a Boolean."

Examples:

- `EntityType EQ Small AND ClaimCount GT 20` → type `Bool`

Rule T-OR (Logical OR)

$$[\text{T-OR}]\ \frac{\Gamma \vdash \phi_1 : \text{Bool} \quad \Gamma \vdash \phi_2 : \text{Bool}}{\Gamma \vdash \phi_1 \text{ OR } \phi_2 : \text{Bool}}$$

Plain English: "ORing two Boolean expressions gives a Boolean."

Category 5: Rounding Functions

Rules T-ROUND, T-FLOOR, T-CEIL

$$[\text{T-ROUND}]\ \frac{\Gamma \vdash e : \tau \quad \tau \in \{\text{Num}\} \cup \{\text{Amt}[c] : c \in \text{ISO-4217}\}}{\Gamma \vdash \text{ROUND}(e) : \tau}$$

(T-FLOOR and T-CEIL have identical structure)

Plain English: "Rounding a value preserves its type. Rounding a Num gives Num. Rounding Amt[EUR] gives Amt[EUR]."

Why this rule exists: Fee calculations often need rounding to comply with official schedules. A key property is **type preservation** - rounding 99.99 EUR doesn't strip away the currency information; you still get EUR.

Examples:

- `ROUND(3.7)` → type `Num` (value: 4)
- `ROUND(99.99<EUR>)` → type `Amt[EUR]` (value: 100 EUR)
- `FLOOR(TotalFee)` where TotalFee : Amt[USD] → type `Amt[USD]`

Category 6: Property Accessors

Rule T-COUNT

$$[\text{T-COUNT}]\ \frac{\Gamma(x) = \text{SymList}}{\Gamma \vdash x!\text{COUNT} : \text{Num}}$$

Plain English: "The !COUNT property of a MULTILIST variable gives the number of selected items as a Num."

Why this rule exists: MULTILISTs let users select multiple options (like multiple countries for designation). !COUNT tells you how many they selected, enabling per-item fee calculations.

Examples:

If `DesignatedCountries` is a MULTILIST where user selected Germany, France, Italy:

- `DesignatedCountries!COUNT` → type `Num` (value: 3)
- `100<EUR> * DesignatedCountries!COUNT` → type `Amt[EUR]` (value: 300 EUR)

Rules T-YEARSTONOW, T-MONTHSTONOW, T-DAYSTONOW, T-MONTHSTONOW-FROMLASTDAY

$$[\text{T-YEARSTONOW}]\ \frac{\Gamma(x) = \text{Date}}{\Gamma \vdash x!\text{YEARSTONOW} : \text{Num}}$$

(Similar structure for the others)

Plain English: "Temporal properties of Date variables give the elapsed time as a dimensionless number."

Property meanings:

- `!YEARSTONOW` - complete years from the date to now
- `!MONTHSTONOW` - complete months from the date to now

- `!DAYSTONOW` - days from the date to now
- `!MONTHSTONOW_FROMLASTDAY` - months from end of the date's month to now

Why these rules exist: IP fee calculations often depend on time elapsed since filing. Maintenance fees increase based on years since grant. Late fees apply after deadlines.

Examples:

- `FilingDate!YEARSTONOW` → type `Num` (e.g., 5 if filed 5 years ago)
- `BaseMaintenance * FilingDate!YEARSTONOW` → type `Amt[EUR]`
- `FilingDate!DAYSTONOW GT 30` → type `Bool` (checking if deadline passed)

Category 7: Set Membership

Rule T-IN

```
        Γ(x) = Sym    Γ(y) = SymList
[T-IN] ──────────────────────────────
            Γ ⊢ x IN y : Bool
```

Plain English: "Testing whether a symbol is IN a symbol list gives a Boolean."

Examples:

- `DE IN DesignatedCountries` → type `Bool`

This enables conditional fees:

```
YIELD 500<EUR> IF DE IN DesignatedCountries
```

Rule T-NIN

```
         Γ(x) = Sym    Γ(y) = SymList
[T-NIN] ──────────────────────────────
            Γ ⊢ x NIN y : Bool
```

Plain English: "Testing whether a symbol is NOT IN a symbol list gives a Boolean."

Category 8: Let Bindings

Rule T-LET

$$[\text{T-LET}]\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \text{LET } x \text{ AS } e_1; e_2 : \tau_2}$$

What the symbols mean:

- $\Gamma \vdash e_1 : \tau_1$ - first, we determine the type of the binding expression
- $\Gamma[x \mapsto \tau_1]$ - then we extend the environment with "x has type τ_1"
- $\Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2$ - under this extended environment, we type the body
- The overall expression has type τ_2

Plain English: "A LET binding introduces a new variable. First type the defining expression, add it to the environment, then type the rest using that extended environment."

Example:

```
LET ClaimFee AS 265<EUR>
LET ExcessClaims AS ClaimCount - 15
YIELD ClaimFee * ExcessClaims
```

Step by step:

1. `265<EUR>` has type `Amt[EUR]`, so `ClaimFee : Amt[EUR]` is added to Γ
2. `ClaimCount - 15` has type `Num`, so `ExcessClaims : Num` is added to Γ
3. `ClaimFee * ExcessClaims` is `Amt[EUR] * Num` → type `Amt[EUR]`

Category 9: Fee Body Typing

These rules ensure all YIELD statements in a fee produce consistent types.

Rule T-YIELD-STMT (Conditional Yield)

$$[\text{T-YIELD-STMT}]\ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \phi : \text{Bool}}{\Gamma \vdash (\text{YIELD } e \text{ IF } \phi) \text{ yields } \tau}$$

What the "yields" judgment means:

- $\Gamma \vdash \text{body yields } \tau$ is a special judgment meaning "all YIELD statements in this body produce type τ"
- It's different from $\Gamma \vdash e : \tau$ (expression typing)

Plain English: "A conditional yield statement yields whatever type its expression has. The condition must be Boolean but doesn't affect the yield type."

Rule T-YIELD-UNCONDITIONAL

```
                      Γ ⊢ e : τ
[T-YIELD-UNCONDITIONAL] ──────────────────
                       Γ ⊢ (YIELD e) yields τ
```

Plain English: "An unconditional yield statement yields the type of its expression."

Rule T-YIELD-SEQ (Sequence of Yields)

```
              Γ ⊢ stmt yields τ    Γ ⊢ rest yields τ
[T-YIELD-SEQ] ──────────────────────────────────────────
                   Γ ⊢ (stmt; rest) yields τ
```

What this rule ensures:

- `stmt` yields τ
- `rest` (the remaining statements) ALSO yields τ
- Therefore the whole sequence yields τ

Plain English: "All yield statements in a fee must produce the SAME type."

Why this rule exists: A fee can't sometimes return EUR and sometimes return USD. Every execution path must produce a consistent type.

Example of violation:

```
COMPUTE FEE BadFee
  YIELD 100<EUR> IF cond1    -- yields Amt[EUR]
  YIELD 200<USD> IF cond2    -- yields Amt[USD] ← TYPE ERROR!
ENDCOMPUTE
```

This fails T-YIELD-SEQ because the two yields have different types.

Rule T-LET-IN-BODY

```
                Γ ⊢ e₁ : τ₁    Γ[x ↦ τ₁] ⊢ rest yields τ₂
[T-LET-IN-BODY] ──────────────────────────────────────────
                   Γ ⊢ (LET x AS e₁; rest) yields τ₂
```

Plain English: "A LET binding in a fee body extends the environment, and the remaining yields are typed under that extended environment."

Category 10: Non-Polymorphic Fees

Rule T-FEE (Basic Fee)

$$[\text{T-FEE}]\ \frac{\Gamma \vdash \text{body yields } \tau \quad \tau \in \{\text{Num}\} \cup \{\text{Amt}[c] : c \in \text{ISO-4217}\}}{\Gamma \vdash (\text{COMPUTE FEE } f \text{ body ENDCOMPUTE}) : \tau}$$

Plain English: "A fee block has whatever type its body yields."

Example:

```
COMPUTE FEE FilingFee
  YIELD 135<EUR>
ENDCOMPUTE
```

The body yields `Amt[EUR]`, so `FilingFee` has type `Amt[EUR]`.

Rule T-FEE-RETURN (Fee with Explicit Return Currency)

$$[\text{T-FEE-RETURN}]\ \frac{\Gamma \vdash \text{body yields Amt}[c] \quad c \in \text{ISO-4217}}{\Gamma \vdash (\text{COMPUTE FEE } f \text{ RETURN } c \text{ body ENDCOMPUTE}) : \text{Amt}[c]}$$

Plain English: "A fee with RETURN c declaration must have a body that yields Amt[c]."

Why this rule exists: The RETURN declaration serves as documentation AND as an extra type check. If someone later changes a yield to use the wrong currency, the type checker will catch it.

Example:

```
COMPUTE FEE FilingFee RETURN EUR
  YIELD 135<EUR>      -- OK: matches declared EUR
ENDCOMPUTE
```

Violation:

```
COMPUTE FEE FilingFee RETURN EUR
  YIELD 135<USD>      -- ERROR: body yields USD but declared EUR
ENDCOMPUTE
```

Category 11: Polymorphic Fees

Rule T-POLY-FEE

```
                α ∉ dom(Γ)    Γ, α : Currency ⊢ body yields Amt[α]
[T-POLY-FEE] ──────────────────────────────────────────────────────
             Γ ⊢ (COMPUTE FEE f<α> RETURN α body ENDCOMPUTE) : ∀α.Amt[α]
```

What the symbols mean:

- `α ∉ dom(Γ)` - α must be a fresh variable not already used
- `Γ, α : Currency` - extend the environment knowing α is some currency
- `body yields Amt[α]` - the body must yield an amount in this unknown currency
- `∀α.Amt[α]` - the fee has polymorphic type "for all currencies α, amount in α"

Plain English: "A polymorphic fee uses a type variable α to represent any currency. The body works with Amt[α], and the fee can be used with ANY specific currency later."

Why this rule exists: Some fee structures are reusable across currencies. A base fee template defined for the European Patent system might be used by Germany (EUR), UK (GBP pre-Brexit), and Switzerland (CHF). Polymorphism lets you define it once.

Example:

```
COMPUTE FEE BaseFee<α> RETURN α
  LET BaseAmount AS 100<α>
  YIELD BaseAmount * ClaimCount
ENDCOMPUTE
```

This fee has type `∀α.Amt[α]` - it works with any currency.

Category 12: Type Instantiation

Rule T-INST

```
          Γ ⊢ f : ∀α.Amt[α]    c ∈ ISO-4217
[T-INST] ──────────────────────────────────
                Γ ⊢ f@c : Amt[c]
```

What the symbols mean:

- `f : ∀α.Amt[α]` - f is a polymorphic fee
- `c ∈ ISO-4217` - c is a specific currency

- `f@c` - instantiate f at currency c
- `Amt[c]` - the result is a concrete amount in currency c

Plain English: "Apply a polymorphic fee to a specific currency to get a concrete fee in that currency."

Why this rule exists: Polymorphic fees are templates. Instantiation "fills in the blank" with a real currency.

Example: If `BaseFee` has type `∀α.Amt[α]`:

- `BaseFee@EUR` has type `Amt[EUR]`
- `BaseFee@USD` has type `Amt[USD]`
- `BaseFee@JPY` has type `Amt[JPY]`

This is how jurisdiction composition works: a child jurisdiction inherits a polymorphic fee from a parent and instantiates it with their local currency.

Part 5: Type Safety - Why This All Matters

5.1 The Guarantee

If an IPFLang program passes type checking, then during execution:

1. **No currency mismatch errors** - You'll never accidentally add EUR to USD
2. **Consistent fee types** - Every fee produces exactly the type it claims
3. **Explicit conversions** - Currency changes only happen at CONVERT calls

5.2 What the Type Checker Prevents

Error	What Would Happen	How Type System Prevents It
`100<EUR> + 50<USD>`	Wrong total (mixing currencies)	T-ADD-AMT requires same currency
`100<EUR> + 50`	Ambiguous (is 50 EUR?)	No rule allows Amt + Num
`100<EUR> - 50`	Ambiguous	No rule allows Amt - Num
`EntityType GT SmallEntity`	Meaningless comparison	T-COMP-ORD excludes Sym
Fee returning EUR sometimes, USD other times	Unpredictable results	T-YIELD-SEQ requires consistent types

5.3 What IS Allowed

Operation	Why It Works
`100<EUR> + 50<EUR>`	Same currency, T-ADD-AMT applies
`100<EUR> * 2`	Scalar multiplication, T-MUL-SCALAR-R applies
`100<EUR> / 4`	Scalar division, T-DIV-AMT-SCALAR applies
`CONVERT(x, EUR, USD) + 50<USD>`	Explicit conversion makes currencies match

Operation	Why It Works
`ROUND(99.99<EUR>)`	Type-preserving operation

Part 6: Putting It All Together - A Complete Example

Let's trace through the type checking of a real fee:

```
COMPUTE FEE ExcessClaimsFee
  LET ClaimFee1 AS 265<EUR>
  LET ClaimFee2 AS 660<EUR>
  CASE ClaimCount LTE 15 AS
    YIELD 0<EUR>
  ENDCASE
  CASE ClaimCount GT 15 AND ClaimCount LTE 50 AS
    YIELD ClaimFee1 * (ClaimCount - 15)
  ENDCASE
  CASE ClaimCount GT 50 AS
    YIELD ClaimFee1 * 35 + ClaimFee2 * (ClaimCount - 50)
  ENDCASE
ENDCOMPUTE
```

Step 1: Initial environment

$$\Gamma_0 = \{ \text{ClaimCount} : \text{Num} \}$$

Step 2: Type the first LET

- `265<EUR>` has type `Amt[EUR]` by T-CURR
- Add to environment: $\Gamma_1 = \Gamma_0[\text{ClaimFee1} \mapsto \text{Amt[EUR]}]$

Step 3: Type the second LET

- `660<EUR>` has type `Amt[EUR]` by T-CURR
- Add to environment: $\Gamma_2 = \Gamma_1[\text{ClaimFee2} \mapsto \text{Amt[EUR]}]$

Step 4: Type the first YIELD

- `0<EUR>` has type `Amt[EUR]` by T-CURR
- First yield produces `Amt[EUR]`

Step 5: Type the second YIELD

- `ClaimCount - 15`:
  - `ClaimCount : Num` (from Γ)
  - `15 : Num` (by T-NUM)
  - By T-SUB-NUM: `ClaimCount - 15 : Num`
- `ClaimFee1 * (ClaimCount - 15)`:

    - `ClaimFee1 : Amt[EUR]`
    - `(ClaimCount - 15) : Num`
    - By T-MUL-SCALAR-R: result is `Amt[EUR]`
- Second yield produces `Amt[EUR]` ✓ (matches first)

Step 6: Type the third YIELD

- `ClaimFee1 * 35`:
    - `ClaimFee1 : Amt[EUR]`, `35 : Num`
    - By T-MUL-SCALAR-R: `Amt[EUR]`
- `ClaimCount - 50`: by T-SUB-NUM: `Num`
- `ClaimFee2 * (ClaimCount - 50)`:
    - `ClaimFee2 : Amt[EUR]`, `(ClaimCount - 50) : Num`
    - By T-MUL-SCALAR-R: `Amt[EUR]`
- `ClaimFee1 * 35 + ClaimFee2 * (ClaimCount - 50)`:
    - Both operands: `Amt[EUR]`
    - By T-ADD-AMT: `Amt[EUR]`
- Third yield produces `Amt[EUR]` ✓ (matches others)

Step 7: Apply T-YIELD-SEQ

- All yields produce `Amt[EUR]`
- By T-YIELD-SEQ: body yields `Amt[EUR]`

Step 8: Apply T-FEE

- Body yields `Amt[EUR]`
- `Amt[EUR]` is valid (it's in {Amt[c] : c ∈ ISO-4217})
- Fee `ExcessClaimsFee` has type `Amt[EUR]`

Result: The fee is well-typed and will always produce a EUR amount.

Glossary

Term	Definition
Axiom	A rule with no premises; always true
Dimensionless	Having no currency unit (just a number)
Environment (Γ)	A mapping from variable names to their types
Inference rule	A logical rule with premises (above line) and conclusion (below line)
Instantiation	Replacing a type variable with a concrete type
Judgment	A formal statement about types (like Γ ⊢ e : τ)
Polymorphic	Working with multiple types via type variables
Premise	A condition that must hold for a rule to apply

Term	Definition
Type scheme	A type that may contain type variables (∀α. ...)
Type variable	A placeholder for an unknown type (like α)
Well-typed	Passing all type-checking rules

Summary

The IPFLang type system ensures **currency safety** through a small set of principles:

1. **Every value has a type** - Either dimensionless (Num, Bool, etc.) or currency-denominated (Amt[c])
2. **Operations preserve or require matching currencies**
    - Addition/subtraction: same currency required
    - Multiplication/division with scalar: currency preserved
    - No mixing dimensionless with dimensioned in addition/subtraction
3. **Conversions are explicit** - CONVERT is the only way to change currencies
4. **Fees must be consistent** - All execution paths produce the same type
5. **Polymorphism enables reuse** - Generic fees work with any currency, then get instantiated

By learning to read these rules, you can:

- Understand why certain code is accepted or rejected
- Design fee calculations that are provably correct
- Debug type errors by tracing which rule failed
- Appreciate the formal foundations of currency safety