プログラミング言語処理系
MiniMLの型推論

末永 幸平

この資料について

- 京都大学工学部専門科目「プログラミング言語処理系」の講義資料
- 講義 Web ページ:
  - https://kuis-isle3sw.github.io/IoPLMaterials/
- 講義をする人: 末永幸平
  - https://researchmap.jp/ksuenaga/
  - https://twitter.com/ksuenaga

アウトライン

- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

ある卒論生の悲劇

- ？？（23:00）「明日の12:00論文締め切りなのに，追加の実験がどうしても必要になった．．．よし，プログラムを書こう．」
- ？？（3:00）「ヨシ！できた！」
- ？？「明日の朝には結果が出るはずだ．プログラムを走らせて寝て明日の朝論文を出そう．」
- （睡眠中）
- ？？（11:00）「いかん，寝過ごすところだった．結果は出たかな．」

```
let rec f x n =
  if n <= 0 then 3 4 (* (A) *)
  else
    let x = (* めっちゃ重い計算 *)
    f x (n + -1)
in
f 0 100000000
```

```
3 4 を評価しようとしましたが,3 は関数ではありません.落ちます.
```

- ？？「」

何が悪かったのか

- ループの回数を減らしてテストをしていなかった
  - そりゃそうだが午前 3:00 にテストがちゃんとできるもんだろうか
- 締め切り直前まで実装をするのが悪い
  - そんな状況になったことの無い者だけが石を投げなさい
- 静的型検査・型推論のない言語を使っていたのが悪い
  - **激しく同意**

本当は怖いプログラムのバグ

- **アリアン 5 ロケット爆発事故 (1996)**
  - プログラムの誤りでロケットが爆発
  - 5億ドルの積み荷が失われる
- **放射線治療用機器 Therac-25 事故 (1985〜1987)**
  - 意図しない量の放射線を浴びて5人が死亡
- **暗号通信用ソフトウェア OpenSSL のバグ (Heartbleed; 2014)**
  - インターネット上のサーバの 66% に影響
- **暗号通貨 Ethereum の流失事件 (The DAO Attack; 2016)**
  - プログラム（スマートコントラクト）のバグで $170M 相当の仮想通貨が流失

- **静的検証で取り除けるバグは取り除くほうが良い**

静的検証

- **プログラムの誤りを実行前に検出する手法**
  - 静的型検査・型推論
    - 実行時型エラーという誤りを検出
    - 型をさらに表現力豊かにすることでさらにいろいろな性質が検証できる
    - Coq 等の証明支援系 (proof assistant) の理論的基礎
  - モデル検査 (model checking)
    - プログラムを抽象化して有限状態オートマトン等の有限状態システムに落とし，網羅的に安全性を検証
  - 抽象解釈 (abstract interpretation)
    - 静的検証のためのプログラムの抽象化手法の理論的枠組み

意外と使われている静的検証手法

- **Infer: Facebook 社が開発を進める検証器 [1]**
  - Java, C, C++, Androidアプリ, iOSアプリの検証が可能
  - NullPointerExceptionの可能性やメモリリークの可能性等を検出
  - AWS, Instagram, Mozilla, UBER, Spotify 等が使用
- **Astree: C プログラムのプログラム解析器 [2]**
  - Airbus A340, A380 の制御ソフトウェアや Jules Vernes ATV (宇宙ステーション補給機) の制御ソフトウェアを検証
- **CPAchecker: C プログラムのためのモデル検査器 [3]**
  - モデル検査: プログラムを自動的に抽象化してエラー状態に到達する可能性がないか検査する手法
  - Linux カーネルのバグを多数発見 [4]

[1] Infer: http://fbinfer.com/
[2] The Astree Static Analyzer: http://www.astree.ens.fr/
[3] CPAchecker: The configurable software-verification platform: https://cpachecker.sosy-lab.org/
[4] https://cpachecker.sosy-lab.org/achieve.php

ここまで実装したインタプリタには静的な型検査・型推論がなかった

- let x = 2 in x + 3
  - 5 に評価される
- let f x = x + 1 in f true
  - f がクロージャに束縛されて，f が true に適用され，最終的に true + 1 が評価されるが，実行時に型エラーが起こる
  - OCaml だと**評価の前に静的に型推論**が行われるため
    実行時の型エラーは起こらない

これから数回でやること

- MiniML のための型推論機構の実装

アウトライン

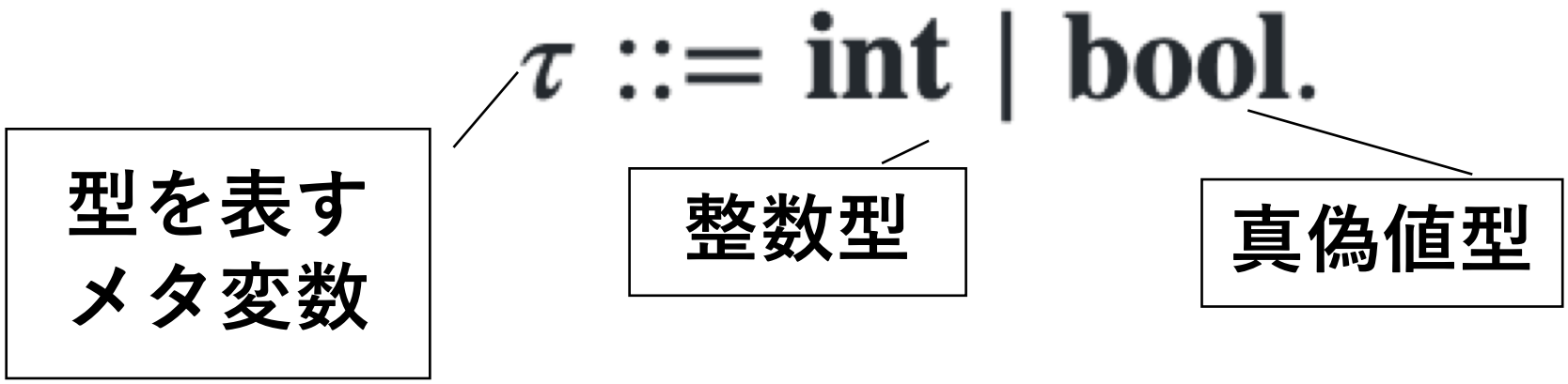
- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

型推論アルゴリズムは通常どうやって設計するか

- まず型システムを定義
  - 型の構文と意味論
  - 「プログラムに型がつく」を意味する関係（**型判断，type judgment**）の定義
  - 「プログラムに型がつけば，プログラムを実行してもエラーが起こらない」という定理（**型健全性, type soundness**）の証明
- 型システムでつく型を推論するアルゴリズムを作る

まずは「型」を定義

MiniML2 には関数がなく，整数と真偽値しかないので int と bool のみが型としてあればよい



「型がつく」をどう定義するか

- **型判断 (type judgment)** と呼ばれる数学的な関係を定義

「式 e に型 τ がつく」を表す型判断 (仮)

$$e : \tau$$

これでいい?

ダメ: e が変数であったときに型が決まらない

型環境 (type environment)

- 変数の型を表すデータ構造

$$\Gamma ::= x_1 : \tau_1, \cdots, x_n : \tau_n$$

型環境を表すメタ変数

x_1の型はτ_1, …, x_nの型はτ_nであることを表す型環境

- Γ に $x:\tau$ が入っているときに，$\Gamma(x) = \tau$ とかく
- Γ 中の $x:\tau$ は自由に順序を入れ替えて良いことにする

型環境のついた型判断

「型環境 Γ のもとで式 e に型 τ がつく」
を表す型判断

$$\Gamma \vdash e : \tau$$

TeX で ¥vdash で出る記号（日本語だと「ト記号」という人もいる）

型付け可能，型付け不能，型推論，型検査

- 式 e が**型付け可能 (well-typed)** とは，$\Gamma \vdash e : \tau$ になるような Γ と τ が存在すること
- どのような Γ と τ を持ってきても $\Gamma \vdash e : \tau$ **とはならない**ときに e は**型付け不能 (ill-typed)** という

- **型推論 (type inference)** とは，以下の問題である
  - 入力: Γ と e
  - 出力: e が型付け可能であれば $\Gamma \vdash e : \tau$ であるような τ を，そうでなければエラー
- **型検査 (type checking)** とは，以下の問題である
  - 入力: Γ と e と τ
  - 出力: $\Gamma \vdash e : \tau$ であるかどうか

アウトライン

- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

正しい型判断

- 型判断には正しいものと誤ったものがある

$$\emptyset \vdash 3 : int$$ — 正しい

$$x{:}int \vdash x+5 : int$$ — 正しい

$$x{:}bool \vdash \text{if } x \text{ then } 3 \text{ else } 5 : int$$ — 正しい

$$x{:}int \vdash \text{if } x \text{ then } 3 \text{ else } 5 : int$$ — 誤り

正しい型判断の定義の仕方

- 「正しい推論の形」を推論規則を使って定義し
  それらで導ける型判断のみを正しい型判断とする

横棒の上にある
条件がすべて
正しいならば

$$\frac{\Gamma \vdash e_1 : \mathrm{int} \qquad \Gamma \vdash e_2 : \mathrm{int}}{\Gamma \vdash e_1 + e_2 : \mathrm{int}}\ (\text{T-Plus})$$

規則の名前

横棒の下にある型判断
は正しい

正しい型判断の定義の仕方

- **「正しい推論の形」を推論規則を使って定義し**
  それらで導ける型判断のみを正しい型判断とする

横棒の上には条件がないので

$$\frac{}{\Gamma \vdash n : \mathrm{int}}\ (\text{T-Int})$$

横棒の下にある型判断は無条件に正しい

正しい型判断の定義の仕方

- **「正しい推論の形」を推論規則を使って定義し**
  それらで導ける型判断のみを正しい型判断とする

横棒の上には付帯条件が来ることもある

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{(T-Var)}$$

型判断の正しさの示し方

- 推論規則中のメタ変数を具体
  型判断を導くための十分条件

$$\Gamma \vdash e_1 : \mathrm{int} \qquad \Gamma \vdash e_2 : \mathrm{int}$$

前提中のこれらの型判断が
すべて正しいことを示せばよい

$$\frac{x{:}\mathrm{int} \vdash x : \mathrm{int} \qquad x{:}\mathrm{int} \vdash 5 : \mathrm{int}}{x{:}\mathrm{int} \vdash x+5 : \mathrm{int}}\ (\text{T-Plus})$$

この型判断が正しいことを
示すには

型判断の正しさの示し方

- 推論規則中のメタ変数を具体化すると型判断を導くための十分条件がわかる

$$\frac{}{\Gamma \vdash n : \mathbf{int}}\ (\text{T-Int})$$

$$\frac{\dfrac{}{x{:}\mathbf{int} \vdash x : \mathbf{int}} \quad \dfrac{}{x{:}\mathbf{int} \vdash 5 : \mathbf{int}}\ (\text{T-Int})}{x{:}\mathbf{int} \vdash x{+}5 : \mathbf{int}}\ (\text{T-Plus})$$

型判断の正しさの示し方

- 推論規則中のメタ変数を具体化すると型判断を導くための十分条件がわかる

$$\frac{\Gamma(x)=\tau}{\Gamma \vdash x : \tau}\ \text{(T-Var)}$$

$$\frac{\dfrac{\text{x:intが型環境中にある}}{\text{x:int} \vdash x : \text{int}}\ \text{(T-Var)} \qquad \dfrac{}{\text{x:int} \vdash 5 : \text{int}}\ \text{(T-Int)}}{\text{x:int} \vdash x+5 : \text{int}}\ \text{(T-Plus)}$$

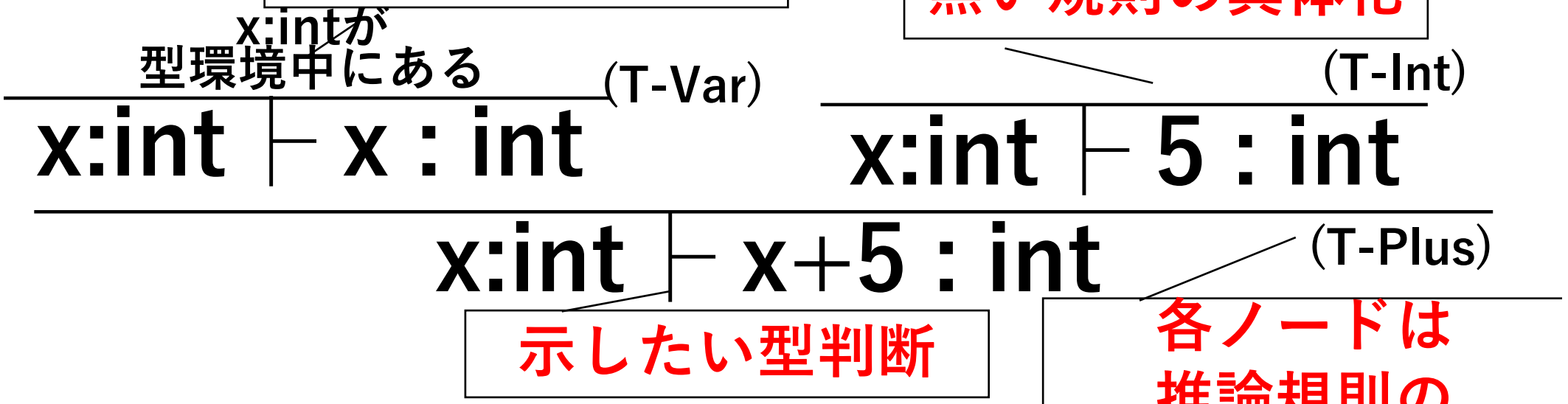
導出木（derivation tree）

- 示したい型
  葉が正しい

推論規
判断

葉が成り立つ
付帯条件

葉が前提の
無い規則の具体化



示したい型判断

各ノードは
推論規則の
具体化

与えられた型判断の正しさを示すには，その
型判断を根とする導出木を一つ作ればよい

アウトライン

- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

変数に関する規則

型環境 Γ に x の型が τ と書いてあれば

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{(T-Var)}$$

Γ の下で x の型が τ であるという型判断を導いてよい

整数定数に関する規則

無条件に任意の整数定数
n について

$$\frac{}{\Gamma \vdash n : \mathsf{Int}}\ (\text{T-Int})$$

n が Int 型であるという
型判断を導いてよい

真偽値定数に関する規則

無条件に任意の
真偽値定数 b について

$$\frac{}{\Gamma \vdash b : \mathrm{Bool}}\ (\text{T-Bool})$$

b が Bool 型であるという
型判断を導いてよい

加算演算に関する規則

Γ の下で e_1 も e_2 も Int 型であるという型判断を導けたならば

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash e_1 + e_2 : \mathrm{Int}}\ (\text{T-Plus})$$

式 e_1+e_2 も Int 型であるという型判断を導いてよい

乗算演算に関する規則

Γ の下で e_1 も e_2 も Int 型であるという型判断を導けたならば

$$\frac{\Gamma \vdash e_1 : \mathbf{Int} \qquad \Gamma \vdash e_2 : \mathbf{Int}}{\Gamma \vdash e_1 * e_2 : \mathbf{Int}}\ (\text{T-Mult})$$

式 e_1*e_2 も Int 型であるという型判断を導いてよい

比較演算に関する規則

Γ の下で e_1 も e_2 も Int 型であるという型判断を導けたならば

$$\frac{\Gamma \vdash e_1 : \mathbf{Int} \qquad \Gamma \vdash e_2 : \mathbf{Int}}{\Gamma \vdash e_1 < e_2 : \mathbf{Bool}} \text{(T-Mult)}$$

式 $e_1 < e_2$ が Bool 型であるという判断を導いてよい

if 式に関する規則

Γ の下で e_1 と e_2 が同じ型 τ であるという型判断を導けて

Γ の下で e が Bool型であるという型判断を導けて

$$\frac{\Gamma \vdash e : \mathbf{Bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}\ (\text{T-If})$$

式 if e then e_1 else e_2 が τ 型であるという判断を導いてよい

T-If に関する注意

- **T-If は保守的な規則になっている**
  - 実行時型エラーが起こらないのに型付け不能なプログラムが存在する

    例: (if true then 3 else true) + 5
- **保守的にする理由: 自動で型推論をするため**
  - 例えば，(if M then 3 else true) + 5 が実行時型エラーを起こさないことを保証するには，M がいかなる場合にも false に評価されないことを証明せねばならず，問題がめちゃくちゃ難しくなる
  - T-If のように「then 節と else 節の型が同じになる」という制約を課せば型付け可能なプログラムが減る代わりに型推論が楽

let 式に関する規則

Γ の下で e_1 が τ_1 型であるという判断が導けて

Γ に x が τ_1 型であるという情報を加えた型環境の下で e_2 が τ_2 型であるという判断が導ければ

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \text{(T-Let)}$$

この式が τ_2 型であるという判断を導いてよい

型環境 Γ, x:τ の定義

- 直観的には
  - Γ に x:τ を追加して得られる型環境
  - 例: Γ=y:Int で τ=Bool なら Γ, x:τ = y:int, x:Bool
  - 例: Γ=∅ で τ=Bool なら Γ, x:τ = x:Bool
- 正式な定義

変数 y が束縛されている型は...

y=x なら x の型 τ で，そうでなければ Γ 中での y の型

$$(\Gamma, x:\tau)(y) = \begin{cases} \tau & (if\ x = y) \\ \Gamma(y) & (otherwise) \end{cases}$$

で定義される．ただし，この型環境の定義域 $\mathbf{Dom}(\Gamma, x:\tau)$ は

$$\mathbf{Dom}(\Gamma, x:\tau) = \mathbf{Dom}(\Gamma) \cup \{x\}$$

で定義される．

ただし，全体としての定義域は Γ の定義域に x を加えた集合

アウトライン

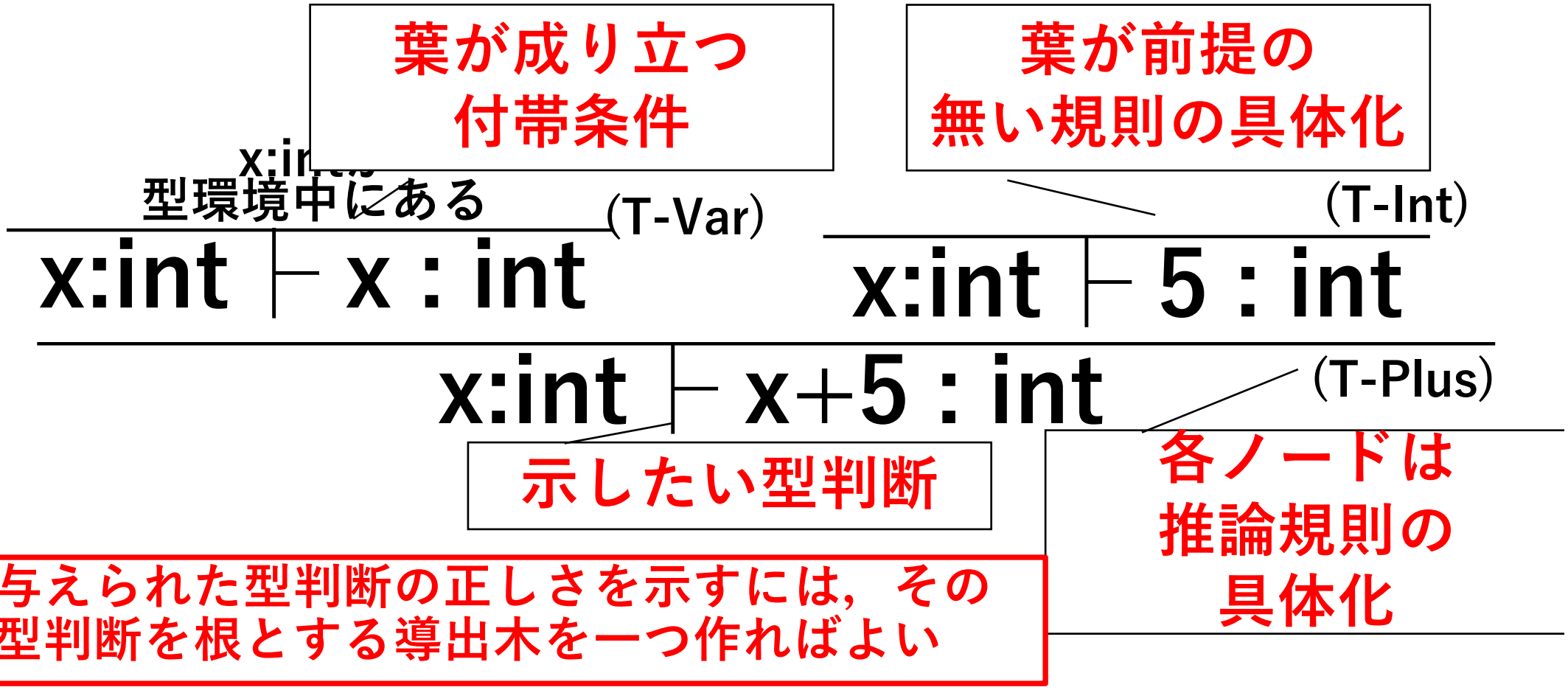
- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

復習

- **型推論（type inference）**とは，以下の問題
  - 入力: Γとe
  - 出力: eが型付け可能であれば$\Gamma \vdash e : \tau$であるようなτを，そうでなければエラー

復習: 導出木

示したい型判断が根に，各ノードが推論規則の具体化になっていて，葉が正しい付帯条件か前提の無い型判断になっている木



ほとんどすべての型システムにおける型推論の基本的戦略

- **目標とする型判断を根として持つ導出木の構築を試みる**

  入力された型環境 Γ と式 e を使って Γ ⊢ e : ? の形に書ける型判断

  - 構築できれば，導出木から？の場所に入るべき型が分かる

どうやったら導出木が構築できるか

- 型付け規則が**構文主導 (syntax-directed)** になっていることを使う

**適用できる規則が e の形から
一意に決まる規則**

- 例: **x:int ⊢ x+5 : ?** を導くための導出木の構築

x:int ⊢ x+5 : ?

どうやったら導出木が構築できるか

- 型付け規則が**構文主導 (syntax-directed)** になっていることを使う

適用できる規則が e の形から
一意に決まる規則

- 例: **x:int ⊢ x+5 : ?** を導くための導出木の構築

式が足し算の形をしているので,
この判断を導くために使えるのは T-Plus のみ

$$\mathbf{x{:}int \vdash x+5 : ?}$$

どうやったら導出木が構築できるか

- 型付け規則が**構文主導 (syntax-directed)** になっていることを使う

適用できる規則が e の形から
一意に決まる規則

- 例: **x:int ⊢ x+5 : ?** を導くための導出木の構築

$$\frac{\mathbf{x{:}int \vdash x : ?} \qquad \mathbf{x{:}int \vdash 5 : ?}}{\mathbf{x{:}int \vdash x+5 : ?}}\ \text{T-Plus}$$

どうやったら導出木が構築できるか

- 型付け規則が**構文主導 (syntax-directed)** になっていることを使う

適用できる規則が e の形から
一意に決まる規則

- 例: **x:int ⊢ x+5 : ?** を導くための導出木の構築

式が変数なので,
使えるのは T-Var のみ

$$\frac{\mathbf{x{:}int} \vdash \mathbf{x : ?} \qquad \mathbf{x{:}int} \vdash \mathbf{5 : ?}}{\mathbf{x{:}int} \vdash \mathbf{x+5 : ?}}\ \text{T-Plus}$$

どうやったら導出木が構築できるか

- 型付け規則が**構文主導 (syntax-directed)** になっていることを使う

適用できる規則が e の形から
一意に決まる規則

- 例: **x:int ⊢ x+5 : ?** を導くための導出木の構築

付帯条件は
確かに成り立っている

$$\dfrac{\dfrac{x{:}int \in (x{:}int)}{x{:}int \vdash x : ?}\ \text{T-Var} \qquad x{:}int \vdash 5 : ?}{x{:}int \vdash x+5 : ?}\ \text{T-Plus}$$

どうやったら導出木が構築できるか

- 型付け規則が**構文主導 (syntax-directed)** になっていることを使う

適用できる規則が e の形から
一意に決まる規則

- 例: **x:int ⊢ x+5 : ?** を導くための導出木の構築

式が整数リテラルなので
使えるのは T-Int のみ

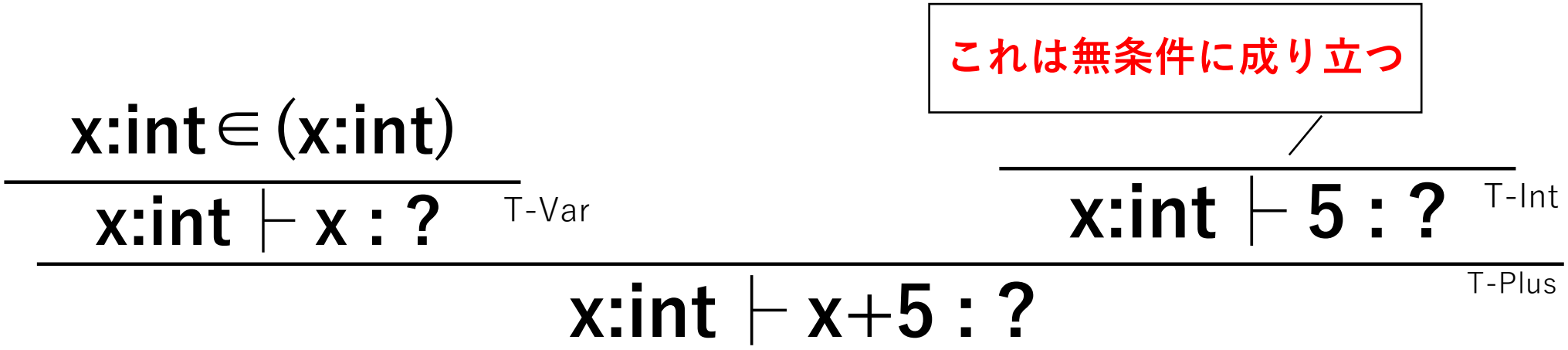
$$\dfrac{\dfrac{x{:}int \in (x{:}int)}{x{:}int \vdash x : ?}\ \text{T-Var} \qquad x{:}int \vdash 5 : ?}{x{:}int \vdash x+5 : ?}\ \text{T-Plus}$$

どうやったら導出木が構築できるか

- 型付け規則が**構文主導 (syntax-directed)** になっていることを使う

適用できる規則が e の形から
一意に決まる規則

- 例: **x:int ⊢ x+5 : ?** を導くための導出木の構築

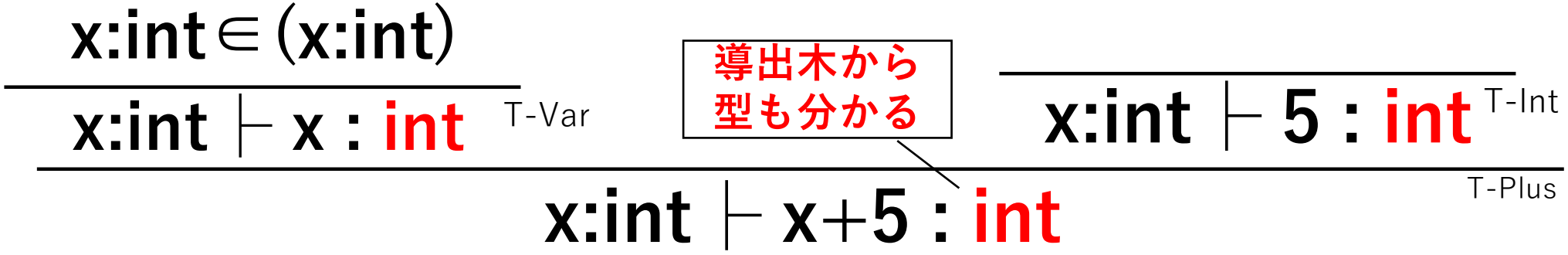


どうやったら導出木が構築できるか

- 型付け規則が**構文主導 (syntax-directed)** になっていることを使う

適用できる規則が e の形から
一意に決まる規則

- 例: **x:int ⊢ x+5 : ?** を導くための導出木の構築



アルゴリズムとして書き下す際のアイデア

- 式の形から適用すべき規則を決め，**その規則を下から上に読む**
  - 具体的には: 型環境 Γ と式 e を受け取り，e の形で場合分け
    - e が足し算 e_1+e_2 であった場合
      - Γ と e_1 を入力として受け取り，型 τ_1 を得る
      - Γ と e_2 を入力として受け取り，型 τ_2 を得る
      - 型 τ_1 と τ_2 がそれぞれ int であることを確認し，型 int を返す
    - …

- 具体的なソースコード: 教科書で説明

アウトライン

- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

型システムの MiniML3 への拡張のレシピ

- 型の拡張
  - 関数型の導入
- 型付け規則の拡張
  - 関数抽象・関数適用のための規則

型の拡張

- 関数値が出てくるので，型の構文に関数型を導入

$$\tau ::= \mathbf{int} \mid \mathbf{bool} \mid \tau_1 \rightarrow \tau_2$$

τ_1型の値を受け取って
τ_2型の値を返す
関数の型

fun x → e のための規則

型環境 $\Gamma, x{:}\tau_1$ の下でeの型がτ_2で
あるという判断が導ければ

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2}\ \text{(T-Fun)}$$

Γ の下で fun x → e の型が
$\tau_1 \rightarrow \tau_2$ という判断を導いて良い

$e_1\ e_2$ のための規則

型環境 Γ の下でe_1の型が$\tau_1 \to \tau_2$であるという判断が導けて

型環境 Γ の下でe_2の型がτ_1であるという判断が導ければ

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{(T-App)}$$

Γ の下で $e_1\ e_2$ の型が τ_2 という判断を導いて良い

アウトライン

- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

復習: 型推論アルゴリズムは型付け規則を「下から上に」読むことで得られる

- 具体的には: 型環境 Γ と式 e を受け取り，e の形で場合分け
  - e が足し算 e_1+e_2 であった場合
    - Γ と e_1 を入力として受け取り，型 τ_1 を得る
    - Γ と e_2 を入力として受け取り，型 τ_2 を得る
    - 型 τ_1 と τ_2 がそれぞれ int であることを確認し，型 int を返す

$$\frac{\Gamma \vdash e_1 : \mathbf{Int} \qquad \Gamma \vdash e_2 : \mathbf{Int}}{\Gamma \vdash e_1 + e_2 : \mathbf{Int}} \text{(T-Plus)}$$

MiniML3 でも同様?

- fun x → e の場合
  - Γ, x:τ_1 と e を入力として受け取り，型 τ_2 を得る
  - $\tau_1 \rightarrow \tau_2$ を返す

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2}\ \text{(T-Fun)}$$

何かおかしいぞ

**このτ1は今から計算する型の一部なのに，
あらかじめ知っていることを
要求されている！**

- fun x → e の場合
  - Γ, x:τ1 と e を入力として受け取り，型 τ2 を得る
  - τ1→ τ2 を返す

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2}\ (\text{T-Fun})$$

MiniML2 までの型推論アルゴリズムの問題点

- 「まだ分からない型」を再帰呼び出しで
  次の入力として渡さなければならない
  - fun x → e の場合
    - Γ, x:τ_1 と e を入力として受け取り，型 τ_2 を得る
    - $\tau_1 \to \tau_2$ を返す

このτ_1は「まだ分からない」型なのに
e の型推論をする際に渡さなければならない

アウトライン

- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

解決策

「まだ分からない型」を表す変数

- 「まだ分からない型」を表す**型変数**を型に含める
- 未定な型を型変数において，型推論を進める
  - fun x → e の場合
    - Γ, x:α_1 と e を入力として受け取り，型 τ_2 を得る
    - $\alpha_1 \rightarrow \tau_2$ を返す

返される型は
未定な型を表す変数を含みうる

解決策

- 型推論の結果，各型変数がどのような型であると分かったかを表すデータ構造（**型代入**）を返す
  - fun x → e の場合
    - Γ, x:α_1 と e を入力として受け取り，型 τ_2 と**型代入** θ を得る
    - 型代入 θ を $\alpha_1 \rightarrow \tau_2$ に適用した結果と型代入 θ を返す
  - 型代入は型推論の過程で課せられる**制約**を解いて得られる

e を型推論した結果
τ_2 中の型変数がどのような型になったかを表すデータ構造

θ の情報に従って
各型変数に型を代入して
得られる型

例えばこのように型推論を実行したい

- 例: fun x → x + 3
  - fun x → x + 3 を型環境 ∅ の下で型推論する
    - x + 3 を x : **α** の下で型推論する
      - x を x : α の下で型推論して，型 α と型代入 ∅ を得る
      - 3 を x : α の下で型推論して，型 int と型代入 ∅ を得る
      - どちらの型も int でなければならないので， α は int でなければならない
        したがって型 int と型代入 $\theta_1 := \{ \alpha = \text{int} \}$ を返す
    - 全体としては型 α →int に θ_1 を適用して得られる型 int→int と
      型代入 θ_1 を返す

例えばこのように型推論を実行したい

新しく生成された型変数

- 例: fun x → x + 3
  - fun x → x + 3 を型環境 ∅ の下で型推論する
    - x + 3 を x : **α** の下で型推論する
      - x を x : α の下で型推論して，型 α と型代入 ∅ を得る
      - 3 を x : α の下で型推論して，型 int と型代入 ∅ を得る
      - どちらの型も int でなければならないので， α は int でなければならない
        したがって型 int と型代入 $\theta_1 := \{ \alpha = \text{int} \}$ を返す
    - 全体としては型 α →intに θ_1 を適用して得られる型 int→int と
      型代入 θ_1 を返す

例えばこのように型推論を実行したい

- 例: fun x → x + 3
  - fun x → x + 3 を型環境 ∅ の下で型推論する
    - x + 3 を x : **α** の下で型推論する
      - x を x : α の下で型推論して，型 α と型代入 ∅ を得る
      - 3 を x : α の下で型推論して，型 int と型代入 ∅ を得る
      - どちらの型も int でなければならないので， α は int でなければならない
        したがって型 int と型代入 $\theta_1 := \{ \alpha = \text{int} \}$ を返す
    - 全体としては型 α →int に θ_1 を適用して得られる型 int→int と
      型代入 θ_1 を返す

T-Var から

分かることは無いので
型代入は空

例えばこのように型推論を実行したい

- 例: fun x → x + 3
  - fun x → x + 3 を型環境 ∅ の下で型推論する
    - x + 3 を x : **α** の下で型推論する
      - x を x : α の下で型推論して，型 α と型代入 ∅ を得る
      - 3 を x : α の下で型推論して，型 int と型代入 ∅ を得る
      - どちらの型も int でなければならないので， α は int でなければならない
        したがって型 int と型代入 $\theta_1 := \{ \alpha = \text{int} \}$ を返す
    - 全体としては型 α →int に θ_1 を適用して得られる型 int→int と
      型代入 θ_1 を返す

T-Int から

分かることは無いので
型代入は空

例えばこのように型推論を実行したい

- 例: fun x → x + 3
  - fun x → x + 3 を型環境 ∅ の下で型推論する
    - x + 3 を x : **α** の下で型推論する
      - x を x : α の下で型推論して，型 α と型代入 ∅ を得る
      - 3 を x : α の下で型推論して，型 int と型代入 ∅ を得る
      - どちらの型も int でなければならないので， α は int でなければならない
        したがって型 int と型代入 $\theta_1 := \{ \alpha = \text{int} \}$ を返す
    - 全体としては型 α →int に θ_1 を適用して得られる型 int→int と
      型代入 θ_1 を返す

T-Plus で
課せられる制約

制約を解いて得られる
型代入

例えばこのように型推論を実行したい

- 例: fun x → x + 3
  - fun x → x + 3 を型環境 ∅ の下で型推論する
    - x + 3 を x : **α** の下で型推論する
      - x を x : α の下で型推論して，型 α と型代入 ∅ を得る
      - 3 を x : α の下で型推論して，型 int と型代入 ∅ を得る
      - どちらの型も int でなければならないので， α は int でなければならない
        したがって型 int と型代入 $\theta_1 := \{ \alpha = \text{int} \}$ を返す
    - 全体としては型 α →int に θ_1 を適用して得られる型 int→int と
      型代入 θ_1 を返す

T-Fun から素直に得られる型は α →int

θ_1 を α →int に適用して得られる型

例えばこのように型推論を実行したい

- 例: (fun x → x) 3
  - (fun x → x) 3 を型環境 ∅ の下で型推論する
    - fun x → x を型環境∅の下で型推論する
      - x を型環境 x: α の下で型推論して，型 α と型代入∅を得る
      - よって，型 $\alpha \rightarrow \alpha$ と型代入∅を返す
    - 3 を型環境∅の下で型推論して，型 int と型代入∅を得る
    - $\alpha \rightarrow \alpha$ は新しい型変数 γ を用いて int → γ と書けなければならない
      これを解くと $\theta_1 := \{\alpha = \text{int}, \gamma = \text{int}\}$ が得られる
      返される型は α であるから θ_1 を適用すると int
      よって，型 int と型代入 $\{\alpha = \text{int}, \gamma = \text{int}\}$ を返す

例えばこのように型推論を実行したい

- 例: (fun x → x) 3
  - (fun x → x) 3 を型環境 ∅ の下で型推論する
    - fun x → x を型環境∅の下で型推論する
      - x を型環境 x:α の下で型推論して，型 α と型代入∅を得る
      - よって，型 α→α と型代入∅を返す
    - 3 を型環境∅の下で型推論して，型 int と型代入∅を得る
    - α→α は新しい型変数 γ を用いて int→γ と書けなければならない
      これを解くと θ_1:={α=int, γ=int} が得られる
      返される型は α であるから θ_1を適用すると int
      よって，型 int と型代入 {α=int, γ=int }を返す

ここまでは
さっきと同じ

例えばこのように型推論を実行したい

- 例: (fun x → x) 3
  - (fun x → x) 3 を型環境 ∅ の下で型推論する
    - fun x → x を型環境∅の下で型推論する
      - x を型環境 x:α の下で型推論して，型 α と型代入∅を得る
      - よって，型 α→α と型代入∅を返す
    - 3 を型環境∅の下で型推論して，型 int と型代入∅を得る
    - α→α は新しい型変数 γ を用いて int→γ と書けなければならない
      これを解くと θ_1:={α=int, γ=int} が得られる
      返される型は α であるから θ_1を適用すると int
      よって，型 int と型代入 {α=int, γ=int }を返す

ここは T-Int から来た

例えばこのように型推論を実行したい

- 例: (fun x → x) 3
  - (fun x → x) 3 を型環境 ∅ の下で型推論する
    - fun x → x を型環境∅の下で型推論する
      - x を型環境 x:α の下で型推論して，型 α と型代入∅を得る
      - よって，型 α→α と型代入∅を返す
    - 3 を型環境∅の下で型推論して，型 int と型代入∅を得る
    - α→α は新しい型変数 γ を用いて int→γ と書けなければならない
      これを解くと $\theta_1 := \{\alpha = \text{int}, \gamma = \text{int}\}$ が得られる
      返される型は α であるから θ_1 を適用すると int
      よって，型 int と型代入 {α = int, γ = int} を返す

fun x → x が int 型の値を受け取ることが分かったので，α→α はint→(何か)でなければならない

「何か」を型変数 γ を使って表現していることに注意

例えばこのように型推論を実行したい

- 例: (fun x → x) 3
  - (fun x → x) 3 を型環境 ∅ の下で型推論する
    - fun x → x を型環境∅の下で型推論する
      - x を型環境 x:α の下で型推論して，型 α と型代入∅を得る
      - よって，型 α→α と型代入∅を返す
    - 3 を型環境∅の下で型推論して，型 int と型代入∅を得る
    - α→α は新しい型変数 γ を用いて int→γ と書けなければならない
      これを解くと θ_1:={α=int, γ=int} が得られる
      返される型は α であるから θ_1を適用すると int
      よって，型 int と型代入 {α=int, γ=int }を返す

α→α とint→γ を同一にする型代入を求めた（後述）

例えばこのように型推論を実行したい

- 例: (fun x → x) 3
  - (fun x → x) 3 を型環境 ∅ の下で型推論する
    - fun x → x を型環境∅の下で型推論する
      - x を型環境 x:α の下で型推論して，型 α と型代入∅を得る
      - よって，型 α→α と型代入∅を返す
    - 3 を型環境∅の下で型推論して，型 int と型代入∅を得る
    - α→α は新しい型変数 γ を用いて int →γ と書けなければならない
      これを解くと θ_1:={α=int, γ=int} が得られる
      返される型はαであるから θ_1を適用すると int
      よって，型 int と型代入 {α=int, γ=int }を返す

関数の返り値の型αが全体の型となる

θ_1から返り値の型がintということが分かる

例えばこのように型推論を実行したい

- 例: fun x → fun y → x y
  - fun x → fun y → x y を型環境 ∅ の下で型推論する
    - fun y → x y を型環境x: α の下で型推論する
      - x y を型環境x: α , y: β の下で型推論する
        - x の型は α
        - y の型は β
        - α はある型変数 γ を用いて $\beta \rightarrow \gamma$ と書けなければならない
          この制約を解くと型代入 $\theta 1 := \{\alpha = \beta \rightarrow \gamma\}$を得る
          返り値の型は γ
          よって，型 γ と型代入 θ_1を返す
    - よって，型 $\beta \rightarrow \gamma$ と型代入 θ_1を返す
  - よって，型 $\alpha \rightarrow \beta \rightarrow \gamma$ に θ_1を適用して得られる
    型 $(\beta \rightarrow \gamma) \rightarrow \beta \rightarrow \gamma$ と型代入 θ_1を返す

要するに

- 型推論アルゴリズムの入出力を変更
  - 入力: 型環境 Γ と式 e
  - 出力: 型 τ と型代入 θ

- 型推論の各ステップで，型付け規則から課せられる制約を生成し，その制約を解いて型代入を求める

実装するためにやるべきこと

- 型変数をどう実装するか決める
- 型代入をどう実装するか決める
- 「制約」をどう表現するかを決める
- 型推論の各ステップでどのような制約を
  生成しなくてはならないかを決める
- 制約を解いて型代入を返すアルゴリズムを作る

型変数の実装

型変数の識別子の型
(実体は整数)

```
type tyvar = int (* New! 型変数の識別子を整数で表現 *)

type ty =
    TyInt
  | TyBool
  | TyVar of tyvar (* New! 型変数型を表すコンストラクタ *)
  | TyFun of ty * ty (* New! TFun(t1,t2) は関数型 t1 -> t2 を表す *)
```

型変数を表す型

関数を表す型

- 型変数の識別子と型変数を区別
  - 前者は tyvar 型の値，後者はコンストラクタ TVar を持つ ty 型の値

型変数に関係する関数

```
(* New! 型のための pretty printer (実装せよ) *)
let pp_ty = ...

(* New! 呼び出すたびに,他とかぶらない新しい tyvar 型の値を返す関数 *)
let fresh_tyvar =
  let counter = ref 0 in (* 次に返すべき tyvar 型の値を参照で持っておいて, *)
  let body () =
    let v = !counter in
      counter := v + 1; v (* 呼び出されたら参照をインクリメントして,古い counter の参照先の値を返す *)
  in body

(*  New!  ty に現れる自由な型変数の識別子(つまり,tyvar 型の集合)を返す関数.実装せよ.*)
(* 型は ty -> tyvar MySet.t にすること.MySet モジュールの実装はリポジトリに入っているはず.*)
let rec freevar_ty ty = ...
```

型変数を pretty print する関数

呼び出されるたびに新しい型変数の識別子を返す関数

型に現れる型変数の識別子の集合を MySet.t 型で返す関数

- freevar_ty はあとで使う

型代入の実装

- の前に: そもそも型代入とは何か？

型代入

- 定義域が有限集合である，型変数から型への写像
  - 例: さっきの型推論の説明で出てきた $\{\alpha = \text{int}, \gamma = \text{int}\}$
    - α を int に，γ を int に写像

型変数を与えると型を返す関数

型代入の型への拡張

- 型変数から型への写像である型代入を，
  以下のように型から型への写像，型から型環境への写像に拡張できる

$$\theta\alpha = \begin{cases} \theta(\alpha) & \text{if } \alpha \in \mathbf{dom}(\theta) \\ \alpha & \text{otherwise} \end{cases}$$

$$\theta\mathbf{int} = \mathbf{int}$$

$$\theta\mathbf{bool} = \mathbf{bool}$$

$$\theta(\tau_1 \to \tau_2) = \theta\tau_1 \to \theta\tau_2$$

$$\mathbf{dom}(\theta\Gamma) = \mathbf{dom}(\Gamma)$$

$$(\theta\Gamma)(x) = \theta(\Gamma(x))$$

他の型は準同型的に拡張

型変数は定義に入っていれば代入され，そうでなければ変化しない

型環境は定義域が変わらず各型に代入を作用させて得られる型に

型代入の実装

```
type subst = (tyvar * ty) list
```

型代入を表す OCaml の型

tyvar と ty のペアのリスト

- 意味:
  - 空の型代入 [] は型代入∅（つまり恒等写像＝型を変化させない代入）を表す
  - (id,ty) :: subst の形をした型代入は「id に ty を代入してから，subst の表す型代入を施す」という働きをする型代入を表す

- したがって型代入 [(id1,ty1); (id2,ty2); …; (idn,tyn)] を ty に適用すると
  - ty中のid1にty1を代入して，その後id2にty2を代入して，…，idnにtynを代入して得られる型となる

例

- [(**alpha**, TyInt)] の表す型代入を型 TyFun(**TyVar alpha**, TyBool) に適用すると TyFun(**TyInt**, TyBool) になる
- [(**alpha**, TyInt); (**beta**, TyBool)] の表す型代入を TyFun(**TyVar alpha**, **TyVar beta**) に適用すると TyFun(**TyInt**, **TyBool**) になる

例

- [(alpha, TyVar beta); (beta, TyVar alpha)] の表す型代入を TyFun(TyVar alpha, TyVar beta) に適用すると?
  - まず (alpha, TyVar beta) を TyFun(TyVar alpha, TyVar beta) に適用して TyFun(TyVar beta, TyVar beta) が得られる
  - 次に (beta, TyVar alpha) を TyFun(TyVar beta, TyVar beta) に適用して TyFun(TyVar alpha, TyVar alpha) が得られる

- TyFun(TyVar alpha, TyVar beta) 中の alpha を TyVar beta に, beta を TyVar alpha に**同時に**適用して得られる TyFun(TyVar beta, TyVar alpha) **ではない**ことに注意

アウトライン

- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

型推論を実装するためにやるべきこと

- 型変数をどう実装するか決める
- 型代入をどう実装するか決める
- 「制約」をどう表現するかを決める
- 型推論の各ステップでどのような制約を
  生成しなくてはならないかを決める
- 制約を解いて型代入を返すアルゴリズムを作る

どのような制約が生じうるか

- 型付け規則を眺めながら，どのような制約が出てくるかを見てみよう

加算演算に関する規則から生じる制約

e_1 と e_2 の型がそれぞれ
Int でなければならない

$$\frac{\Gamma \vdash e_1 : \mathbf{Int} \qquad \Gamma \vdash e_2 : \mathbf{Int}}{\Gamma \vdash e_1 + e_2 : \mathbf{Int}} \text{(T-Plus)}$$

if 式に関する規則から生じる制約

e_1 と e_2 の型が同じでなければならない

e の型は Bool でなければならない

$$\frac{\Gamma \vdash e : \mathbf{Bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}\ (\text{T-If})$$

$e_1\ e_2$ のための規則

e_1の型が関数型の形$\tau_1 \to \tau_2$を していなければならない

e_2の型がe_1の引数の 型τ_1と同じで なければならない

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{(T-App)}$$

というわけで，生じうる制約は

- $\tau_1 = \tau_2$ という形の型の等式制約
  - **e_1 と e_2 の型がそれぞれ Int でなければならない**
  - **e_1 と e_2 の型が同じでなければならない**
  - **e の型は Bool でなければならない**
  - **e_1の型が関数型の形$\tau_1 \to \tau_2$をしていなければならない**
  - **e_2の型がe_1の引数の型τ_1と同じでなければならない**

等式制約の表現方法

- $\tau_1 = \tau_2$ という形の型の等式制約
  - **e_1 と e_2 の型がそれぞれ Int でなければならない**
  - **e_1 と e_2 の型が同じでなければならない**
  - **e の型は Bool でなければならない**
  - **e_1の型が関数型の形$\tau_1 \to \tau_2$をしていなければならない**
  - **e_2の型がe_1の引数の型τ_1と同じでなければならない**

等式制約の表現方法

- $\tau_1=\tau_2$ という形の型の等式制約
  - **(e_1 の型)=Int, (e_2 の型)=Int**
  - **(e_1 の型)=(e_2 の型)**
  - **(e の型)=Bool**
  - **(e_1の型)= $\alpha \rightarrow \tau$, (e_2の型)= α**

等式制約の型推論器内での表現

- 制約 $\tau_1 = \tau_2$ をペア (τ_1, τ_2) で表現
- 制約の集合はリスト $[(\tau_{11}, \tau_{21}); (\tau_{12}, \tau_{22}); \cdots; (\tau_{1N}, \tau_{2N})]$ で表現

- 例: $[(\alpha, \beta \to \beta); (\beta, \mathrm{Int})]$ は制約集合 $\{\alpha = \beta \to \beta, \beta = \mathrm{Int}\}$ を表現

アウトライン

- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

実装するためにやるべきこと

- 型変数をどう実装するか決める
- 型代入をどう実装するか決める
- 「制約」をどう表現するかを決める
- 型推論の各ステップでどのような制約を
  生成しなくてはならないかを決める
- 制約を解いて型代入を返すアルゴリズムを作る

欲しいアルゴリズム

- 入力: 等式制約の集合 $\{ \tau_{11} = \tau_{12},\ \tau_{12} = \tau_{22}, \cdots,\ \tau_{1N} = \tau_{2N} \}$
- 出力: すべての等式制約を満たす型代入 θ

$\theta\ \tau_{11} = \theta\ \tau_{12}$ かつ … かつ $\theta\ \tau_{1N} = \theta\ \tau_{2N}$ が成り立つような型代入 θ

単一化 (unification)

- 項の等式制約を解いて代入を得るアルゴリズム
  - 今回の文脈においては「項」は「型」で「代入」は「型代入」
  - 型推論以外にも論理プログラミング言語 Prolog 等の言語処理系の実装でよく使われる

単一化アルゴリズムの概要

- 制約集合 $\{ \tau_{11} = \tau_{12}, \tau_{12} = \tau_{22}, \cdots, \tau_{1N} = \tau_{2N} \}$ から一つ制約 $\tau = \tau'$ を取り出す
- この制約から τ と τ' の形に応じて (1) 分解して新しい制約を作ったり，(2) 型代入を生成したり，(3) エラーを報告したりする
  - 例: $\tau_a \rightarrow \tau_a' = \tau_b \rightarrow \tau_b'$ という形の制約だったら，制約 $\{ \tau_a = \tau_b, \tau_a' = \tau_b' \}$ を生成して制約集合に追加して単一化を続ける
  - 例: そもそも τ と τ' が全く同じ形だったら，これを制約集合から削除して単一化を続ける

アルゴリズムの定義

- **Unify(C): 等式制約集合 C を解いて得られる型代入**

$$
\begin{aligned}
&\mathit{Unify}(\emptyset) && = && \emptyset \\
&\mathit{Unify}(X' \uplus \{\tau = \tau\}) && = && \mathit{Unify}(X') \\
&\mathit{Unify}(X' \uplus \{\tau_{11} \to \tau_{12} = \tau_{21} \to \tau_{22}\}) && = && \mathit{Unify}(\{\tau_{11} = \tau_{21}, \tau_{12} = \tau_{22}\} \uplus X') \\
&\mathit{Unify}(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) && = && \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
&\mathit{Unify}(X' \uplus \{\tau = \alpha\}) \quad (\text{if } \tau \neq \alpha) && = && \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
&\mathit{Unify}(X \uplus \tau_1 = \tau_2) && = && \mathbf{Error} \quad (\text{Otherwise})
\end{aligned}
$$

アルゴリズムの定義

- **Unify(C): 等式制約集合 C を解**

制約集合が空なら空の代入（恒等写像）が返される

$$
\begin{aligned}
&\mathit{Unify}(\emptyset) && = && \emptyset \\
&\mathit{Unify}(X' \uplus \{\tau = \tau\}) && = && \mathit{Unify}(X') \\
&\mathit{Unify}(X' \uplus \{\tau_{11} \to \tau_{12} = \tau_{21} \to \tau_{22}\}) && = && \mathit{Unify}(\{\tau_{11} = \tau_{21}, \tau_{12} = \tau_{22}\} \uplus X') \\
&\mathit{Unify}(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) && = && \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
&\mathit{Unify}(X' \uplus \{\tau = \alpha\}) \quad (\text{if } \tau \neq \alpha) && = && \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
&\mathit{Unify}(X \uplus \tau_1 = \tau_2) && = && \mathbf{Error} \quad (\text{Otherwise})
\end{aligned}
$$

アルゴリズムの定義

- **Unify(C): 等式制約集合 C を解いて得られる型代入**

$\tau = \tau$ という制約は
単に削除

$$
\begin{array}{lcl}
Unify(\emptyset) & & \\
Unify(X' \uplus \{\tau = \tau\}) & = & Unify(X') \\
Unify(X' \uplus \{\tau_{11} \to \tau_{12} = \tau_{21} \to \tau_{22}\}) & = & Unify(\{\tau_{11} = \tau_{21}, \tau_{12} = \tau_{22}\} \uplus X') \\
Unify(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) & = & \begin{cases} Unify([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
Unify(X' \uplus \{\tau = \alpha\}) \quad (\text{if } \tau \neq \alpha) & = & \begin{cases} Unify([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
Unify(X \uplus \tau_1 = \tau_2) & = & \mathbf{Error} \quad (\text{Otherwise})
\end{array}
$$

アルゴリズムの定義

- **Unify(C): 等式制約集合 C を解いて得られる型代入**

関数型同士の間の等式制約は分解

$$
\begin{array}{lll}
\mathit{Unify}(\emptyset) & & \\
\mathit{Unify}(X' \uplus \{\tau = \tau\}) & & \\
\mathit{Unify}(X' \uplus \{\tau_{11} \to \tau_{12} = \tau_{21} \to \tau_{22}\}) & = & \mathit{Unify}(\{\tau_{11} = \tau_{21}, \tau_{12} = \tau_{22}\} \uplus X') \\
\mathit{Unify}(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) & = & \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
\mathit{Unify}(X' \uplus \{\tau = \alpha\}) \quad (\text{if } \tau \neq \alpha) & = & \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
\mathit{Unify}(X \uplus \tau_1 = \tau_2) & = & \mathbf{Error} \quad (\text{Otherwise})
\end{array}
$$

アルゴリズムの定義

- **Unify(C): 等式制約集合 C を解いて得られる型代入**

$$
\begin{aligned}
&\mathit{Unify}(\emptyset) && = && \emptyset \\
&\mathit{Unify}(X' \uplus \{\tau = \tau\}) && && \\
&\mathit{Unify}(X' \uplus \{\tau_{11} \to \tau_{12} = \tau_{21} \to \tau \\
&\mathit{Unify}(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) && = && \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
&\mathit{Unify}(X' \uplus \{\tau = \alpha\}) \quad (\text{if } \tau \neq \alpha) && = && \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
&\mathit{Unify}(X \uplus \tau_1 = \tau_2) && = && \mathbf{Error} \quad (\text{Otherwise})
\end{aligned}
$$

このケースは後述

アルゴリズムの定義

- **Unify(C): 等式制約集合 C を解いて得られる型代入**

$$
\begin{array}{lcl}
\mathit{Unify}(\emptyset) & = & \emptyset \\
\mathit{Unify}(X' \uplus \{\tau = \tau\}) & = & \mathit{Unify}(X') \\
\mathit{Unify}(X' \uplus \{\tau_{11} \to \tau_{12} = \tau_{21} \to \tau_{22}\}) & = & \mathit{Unify}(\{\tau_{11} = \tau_{21}, \tau_{12} = \tau_{22}\} \uplus X') \\
\mathit{Unify}(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) & = & \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
\mathit{Unify}(X' \uplus \{\tau = \alpha\}) \quad (\text{if } \tau \neq \alpha) & = & \begin{cases} \mathit{Unify}([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases} \\
\mathit{Unify}(X \uplus \tau_1 = \tau_2) & = & \mathbf{Error} \quad (\text{Otherwise})
\end{array}
$$

どれにも当てはまらない場合はエラー

$\alpha = \tau$ という制約の単一化

$$Unify(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) \quad = \quad \begin{cases} Unify([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases}$$

- 残りの成約 X' に代入 $[\alpha = \tau]$ を施して得られる $[\alpha = \tau]X'$ を単一化して代入 Unify($[\alpha = \tau]X'$) を得る
- その代入と $[\alpha = \tau]$ を合成
- **ただし，これをやっていいのは τ に α が現れないときだけで，そうでなければエラー**
  - このチェックを **occur check** と呼ぶ

$\alpha = \tau$ という制約の単一化

$$Unify(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) \qquad = \begin{cases} Unify([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases}$$

- 例:
  Unify({ $\alpha = \beta \to \beta$, $\beta = \gamma$, $\delta = \alpha \to$ Int})

まずこれを処理しよう

$\alpha = \tau$ という制約の単一化

$$Unify(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) \quad = \begin{cases} Unify([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases}$$

- 例:
  Unify({ $\alpha = \beta \to \beta$, $\beta = \gamma$, $\delta = \alpha \to$Int}) =
  Unify({ $\beta = \gamma$, $\delta = (\beta \to \beta) \to$Int}) ○ [$\alpha = \beta \to \beta$]

残りの制約の α の部分に $\beta \to \beta$ を代入

[$\alpha = \beta \to \beta$] を右から合成

次にこれを処理

$\alpha = \tau$ という制約の単一化

$$Unify(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) \qquad = \begin{cases} Unify([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases}$$

- 例:
  $\mathrm{Unify}(\{\alpha = \beta \to \beta,\ \beta = \gamma,\ \delta = \alpha \to \mathrm{Int}\}) =$
  $\mathrm{Unify}(\{\beta = \gamma,\ \delta = (\beta \to \beta) \to \mathrm{Int}\}) \circ [\alpha = \beta \to \beta] =$
  $\mathrm{Unify}(\{\delta = (\gamma \to \gamma) \to \mathrm{Int}\}) \circ [\beta = \gamma] \circ [\alpha = \beta \to \beta] =$

βのところに γを代入

$\beta = \gamma$を 右から合成

$\alpha = \tau$ という制約の単一化

$$Unify(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) \qquad = \begin{cases} Unify([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases}$$

- 例:
  $\mathrm{Unify}(\{\alpha = \beta \to \beta,\ \beta = \gamma,\ \delta = \alpha \to \mathrm{Int}\}) =$
  $\mathrm{Unify}(\{\beta = \gamma,\ \delta = (\beta \to \beta) \to \mathrm{Int}\}) \circ [\alpha = \beta \to \beta] =$
  $\mathrm{Unify}(\{\delta = (\gamma \to \gamma) \to \mathrm{Int}\}) \circ [\beta = \gamma] \circ [\alpha = \beta \to \beta] =$
  $\mathrm{Unify}(\varnothing) \circ [\delta = (\gamma \to \gamma) \to \mathrm{Int}] \circ [\beta = \gamma] \circ [\alpha = \beta \to \beta]$

$\delta = (\gamma \to \gamma) \to \mathrm{Int}$ を右から合成

これは恒等写像

$\alpha = \tau$ という制約の単一化

$$Unify(X' \uplus \{\alpha = \tau\}) \quad (\text{if } \tau \neq \alpha) \quad = \quad \begin{cases} Unify([\alpha \mapsto \tau]X') \circ [\alpha \mapsto \tau] & (\alpha \notin \mathbf{FTV}(\tau)) \\ \mathbf{Error} & (\text{Otherwise}) \end{cases}$$

- 例:
  $\mathrm{Unify}(\{\alpha = \beta \to \beta,\ \beta = \gamma,\ \delta = \alpha \to \mathrm{Int}\}) =$
  $\mathrm{Unify}(\{\beta = \gamma,\ \delta = (\beta \to \beta) \to \mathrm{Int}\}) \circ [\alpha = \beta \to \beta] =$
  $\mathrm{Unify}(\{\delta = (\gamma \to \gamma) \to \mathrm{Int}\}) \circ [\beta = \gamma] \circ [\alpha = \beta \to \beta] =$
  $\mathrm{Unify}(\varnothing) \circ [\delta = (\gamma \to \gamma) \to \mathrm{Int}] \circ [\beta = \gamma] \circ [\alpha = \beta \to \beta] =$
  $[\delta = (\gamma \to \gamma) \to \mathrm{Int}] \circ [\beta = \gamma] \circ [\alpha = \beta \to \beta]$

最終的な解

subst 型としては
$[(\alpha, \beta \to \beta); (\beta, \gamma); (\delta, (\gamma \to \gamma) \to \mathrm{Int}]$
で表現される

実際のコードの説明

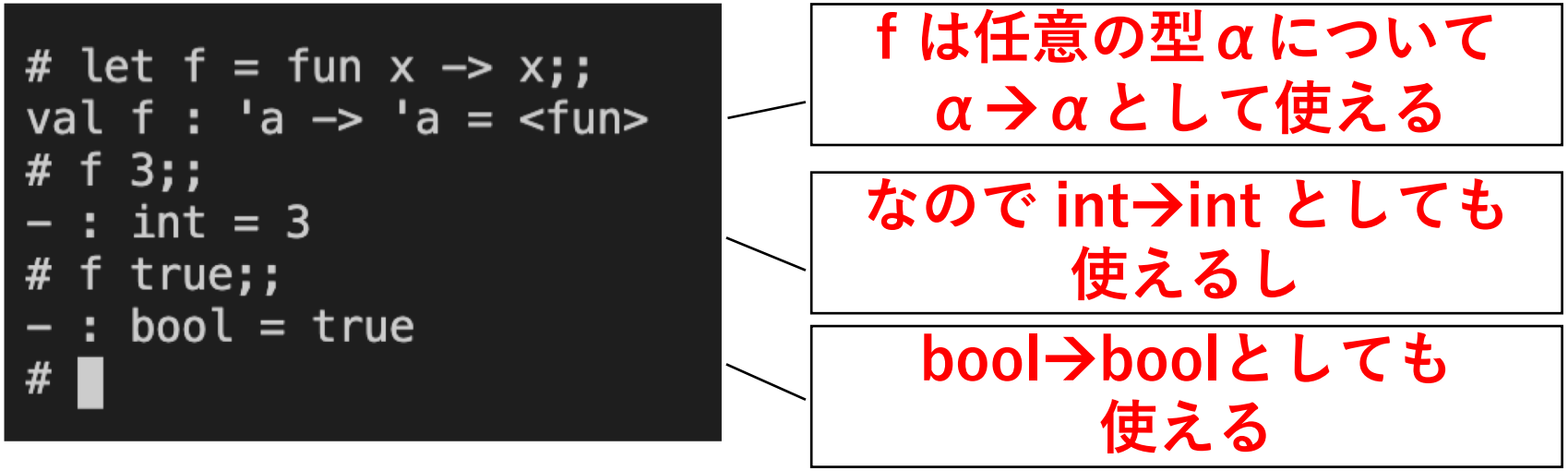
- 教科書で https://kuis-isle3sw.github.io/IoPLMaterials/textbook/chap04-5.html

アウトライン

- プログラム検証へのイントロダクション
- MiniML2 の型システムと型推論
  - 「型システムを定義する」とは
  - 正しい型判断の定め方
  - 導出規則の説明
  - 型推論アルゴリズム設計のストラテジー
- MiniML3 の型推論
  - 型システムの拡張
  - MiniML2 型推論アルゴリズムでのストラテジーの破綻
  - 制約に基づく型推論
  - 単一化による制約解消
- let多相の導入

多相型

- 複数の型として使うことのできる値に与えられる型



- cf. 単相型: 多相でない型

多相型の表現

$$\forall \alpha_1,\ldots,\alpha_n.\ \tau$$

- 任意の $\alpha_1,\ldots,\alpha_n$ について τ として使える型
- 例:
  - fun x → x には型 $\forall \alpha . \alpha \to \alpha$ がつく
    - OCaml では 'a → 'a と表示される
  - fun f → fun x → f x には型 $\forall \alpha, \beta . (\alpha \to \beta) \to \alpha \to \beta$ がつく
    - OCaml では ('a → 'b) → 'a → 'b と表示される

- NB: 実際はこの説明は不正確（後述）

OCaml における多相型のルール

- let や let rec で束縛された変数にのみ多相型がつく (**let多相; let-polymorphism**)
  - 型推論を容易にするために設けられている制限
- その中でも let や let rec による束縛先の式が明らかに値である場合にのみ多相型がつく (**値多相; value restriction**)
  - 参照の存在下で型システムをお手軽に健全にするための制限

OCaml における多相型のルール 1

- let や let rec で束縛された変数にのみ多相型がつく

let で f を束縛

```
# let f = fun x -> x in (f 1, f true);;
- : int * bool = (1, true)
```

f に多相型がつくからこのプログラムは型がつく

fun 式の仮引数として f を束縛

```
# (fun f -> (f 1, f true)) (fun x -> x);;
Error: This expression has type bool but an expression was expected of type
         int
```

f を fun x → x に束縛して (f 1, f true) を評価するという動作は同じなのに,
f に多相型はつかないので型エラー

OCaml における多相型のルール 2

- let や let rec による束縛先の式が明らかに値である場合にのみ型がつく

fun x → x は明らかにクロージャ値

```
# let f = fun x -> x in (f 1, f true);;
- : int * bool = (1, true)
```

f に多相型がつくからこのプログラムは型がつく

関数適用は値ではない

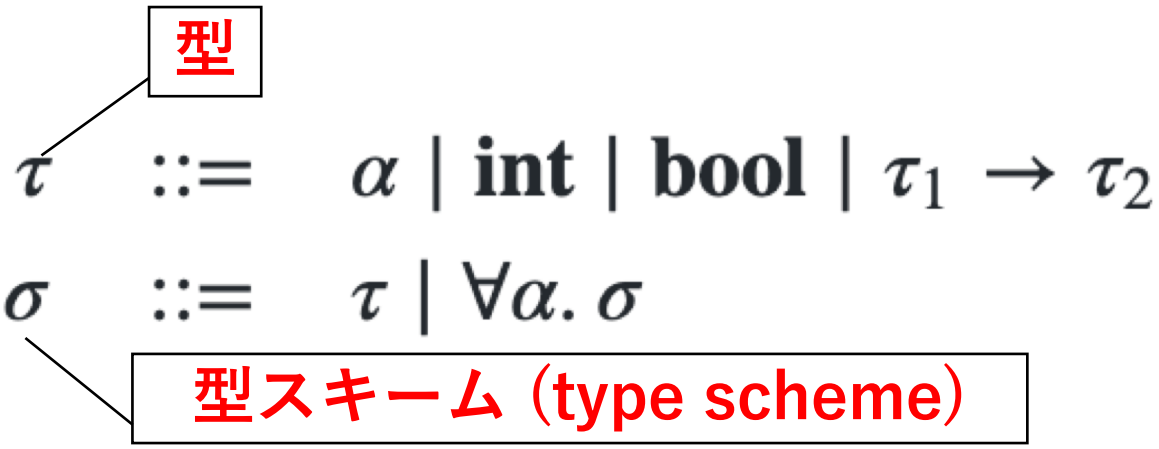
```
# let f = (fun x -> x) (fun x -> x) in (f 1, f true);;
Error: This expression has type bool but an expression was expected of type
         int
```

(fun x → x) (fun x → x) は評価されると (fun x → x) になるのに
f に多相型はつかないので型エラー

多相型での拡張の方針

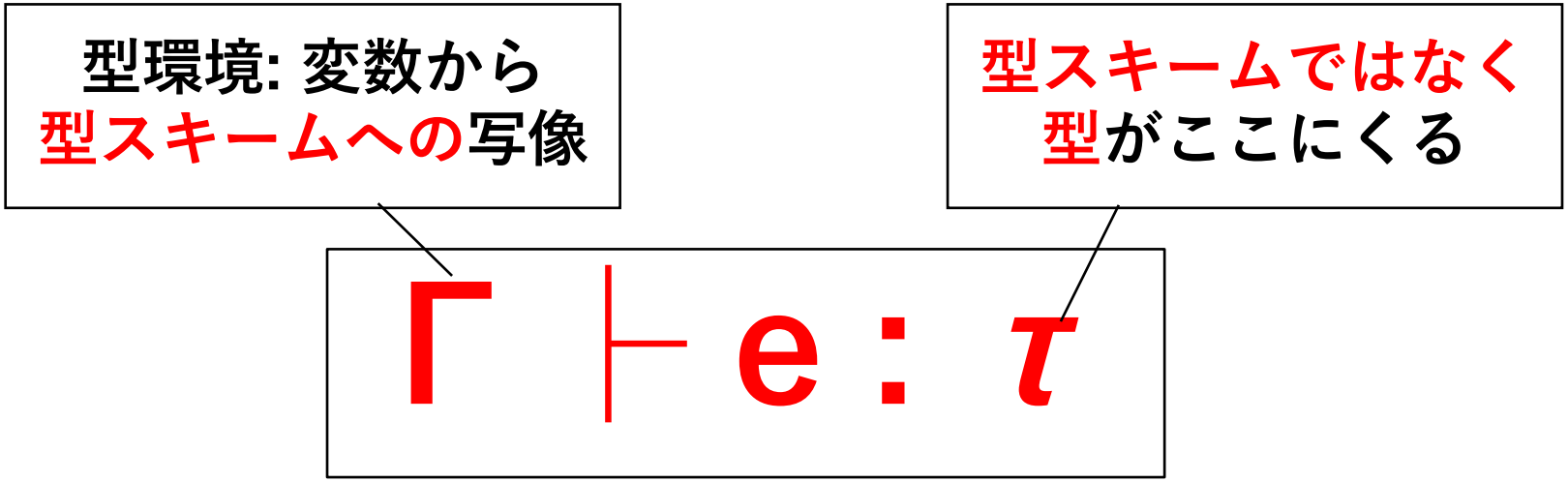
- let x = e in ... の型推論で，e が明らかに値であれば，
  e の推論された型に含まれる型変数のうち
  ∀をつけてよい型変数を決定
- 変数を使う場所で∀のついた型変数を具体的な型に具体化
  - 例: let f = fun x → x in (f 3, f true)
    - 型環境∅の下で fun x → x を型推論して 'a → 'a を得る
    - この場合 'a は∀をつけてよい型変数
      - なぜなのかは後述
    - よって，f は∀'a. 'a→'a を持ち，f:∀'a. 'a→'a の下で (f 3, f true) が型推論される
      - 式 f 3 中の f は 'a を Int に具体化して得られる型 Int → Int が与えられる
      - 式 f true 中の f は 'a を Bool に具体化して得られる型 Bool → Bool が与えられる

型の文法

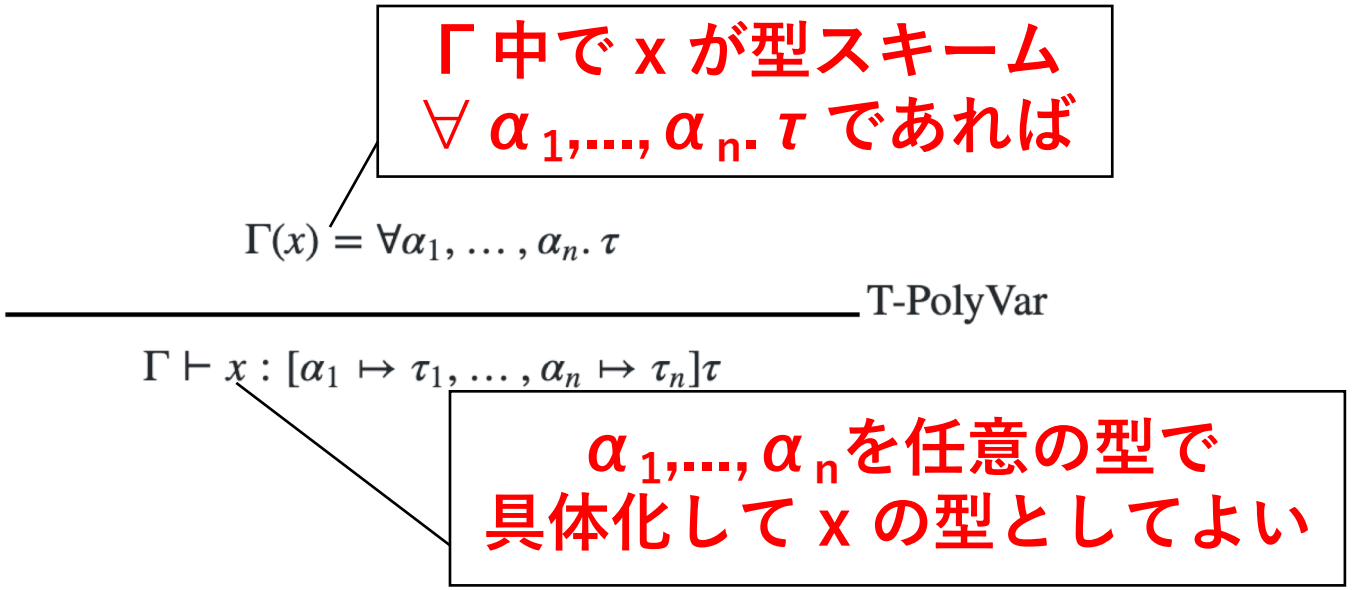


- 具体化された型 τ とは別に，$\forall\, \alpha_1, \ldots, \alpha_n.\ \tau$ を**型スキーム**と呼ぶ
  - 型スキーム中の∀で束縛された型変数を具体化すると型が得られる
  - ∀は τ の外側につくことを強制
    - $(\forall\, \alpha.\alpha \rightarrow \alpha) \rightarrow (\forall\, \alpha.\alpha \rightarrow \alpha)$ のような型は現れない
  - ∀で束縛された型変数が 0 個の場合は通常の型と同じとみなせる
- **型環境は変数から型スキームへの写像に拡張**

型判断の一般形



型付け規則



- 例: さっきの let f = fun x → x in (f 3, f true)
  - 式 (f 3, f true) における型環境は f : ∀ α . α → α だった
  - f 3 中の f の型は， α を Int で具体化した Int → Int としてよい
  - f true 中の f の型は， α を Bool で具体化した Bool → Bool としてよい

let x = e_1 in e_2 のための型付け規則

τ_1 中でこの条件を満たす型変数$\alpha_1,\ldots,\alpha_n$には e_2中の x の型スキームで$\forall$をつけて良い

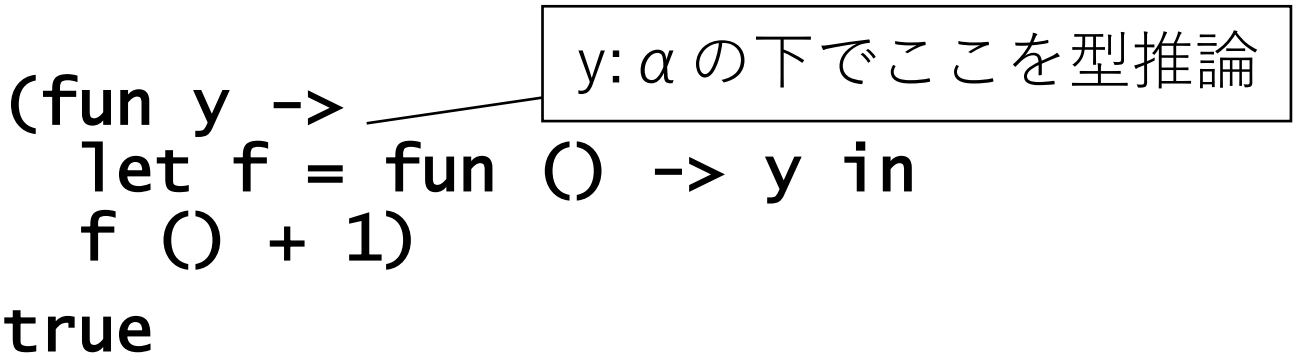
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \forall\alpha_1.\cdots\forall\alpha_n.\tau_1 \vdash e_2 : \tau_2 \quad (\alpha_1,\ldots,\alpha_n \text{ は } \tau_1 \text{ に自由に出現する型変数で } \Gamma \text{ には自由に出現しない})}{\Gamma \vdash \mathbf{let}\ x = e_1 \mathbf{in}\ e_2 : \tau_2}\ \text{T-PolyLet}$$

- なぜ「$\alpha_1,\cdots,\alpha_n$ は τ_1 に自由に出現する型変数で**Γ には自由に出現しない**」という条件が必要か?

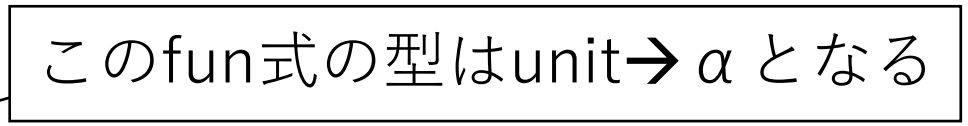
Γ に自由に出現しない型変数にのみ ∀ をつけてよい理由

- Γ に自由に現れる型変数は，**プログラム中の別の場所で生成された制約によって特定の型に単一化されてしまう可能性がある**から

∀を無闇につけると正しくない型システムとなることを示す例



∀を無闇につけると正しくない型システムとなることを示す例




```
(fun y ->
  let f = fun () -> y in
  f () + 1)
true
```

∀を無闇につけると正しくない型システムとなることを示す例

```
(fun y ->
  let f = fun () -> y in
  f () + 1)
true
```

このfun式の型はunit→αとなる

もしαに∀をつけることを許すと
fの型スキームは∀α.unit→αとなる

∀を無闇につけると正しくない型システムとなることを示す例

```
(fun y ->
  let f = fun () -> y in
  f () + 1)
true
```

このfun式の型はunit→αとなる

もしαに∀をつけることを許すと
fの型スキームは∀α.unit→αとなる

すると，αをIntで具体化することが許されてしまうので，f () はInt型となり
f () + 1 にInt型がつく

∀を無闇につけると正しくない型システムとなることを示す例

```
(fun y ->
  let f = fun () -> y in
  f () + 1)
true
```

したがって全体としては
α→Int型がつく

∀を無闇につけると正しくない型システムとなることを示す例

```
(fun y ->
  let f = fun () -> y in
  f () + 1)
true
```

関数適用からα＝Boolという制約が生成され全体としてInt 型が与えられる

∀を無闇につけると正しくない型システムとなることを示す例

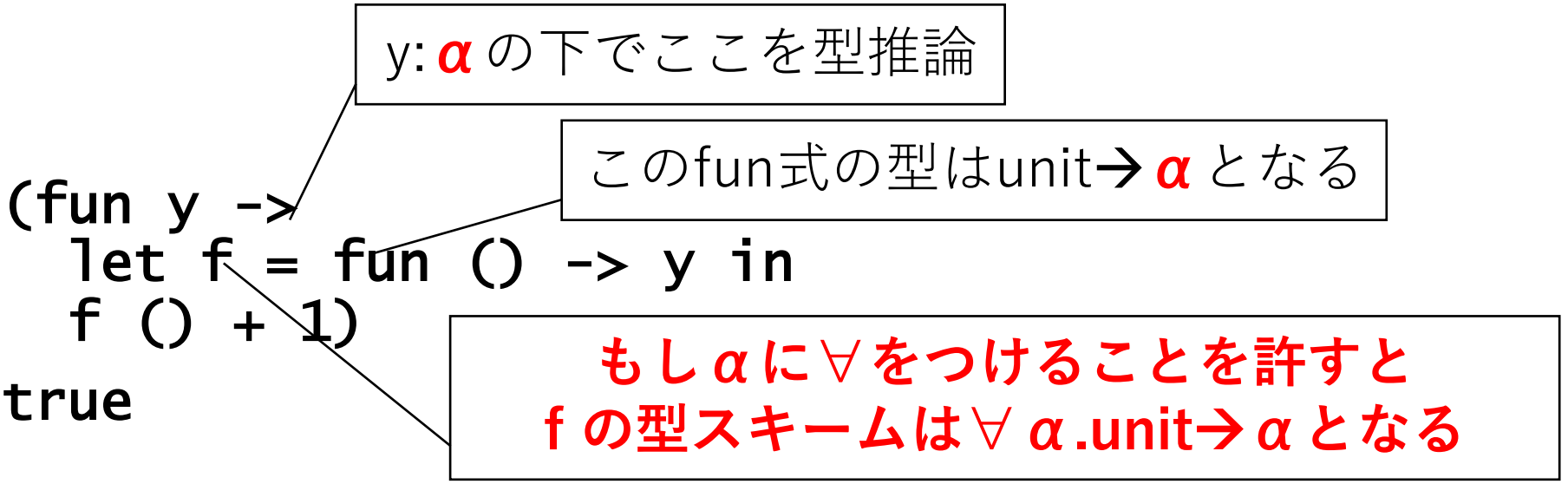
```
(fun y ->
  let f = fun () -> y in
  f () + 1)
true
```

ところが y に true が渡されているので...

f () の返り値は true となり
実行時型エラーが起こる

何が悪かったのか

- fun () → y の型 unit → α は，あとで Int と単一化されるはずの型変数だったので，∀をつけてはいけなかった
  - この時点の型環境は y:α で，α が現れているので，∀をつけてはいけない



というわけで

τ_1 中でこの条件を満たす型変数 $\alpha_1,\ldots,\alpha_n$ には e_2 中の x の型スキームで $\forall$ をつけて良い

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \forall\alpha_1.\cdots\forall\alpha_n.\tau_1 \vdash e_2 : \tau_2 \quad (\alpha_1,\ldots,\alpha_n \text{ は } \tau_1 \text{ に自由に出現する型変数で } \Gamma \text{ には自由に出現しない})}{\Gamma \vdash \mathbf{let}\ x = e_1 \mathbf{in}\ e_2 : \tau_2}\ \text{T-PolyLet}$$

- 「$\alpha_1,\cdots,\alpha_n$ は τ_1 に自由に出現する型変数で **Γ には自由に出現しない**」という条件は，別の型と単一化されうる型変数に $\forall$ をつけることを防ぐために必要

ソースコードの説明

- 教科書を見ながら https://kuis-isle3sw.github.io/IoPLMaterials/textbook/chap04-7.html