プログラミング言語 4
オブジェクト指向と静的型システム
Programming Languages 4
Object-Oriented Languages and Static Type Systems

田浦

目次

Contents

オブジェクト指向言語の特徴(復習)

- 多相性 (異なるデータを扱うコードが一つですむ)
  - プログラムの文面上一つの式や変数が, 実行時には複数の種類の型でありうる (例: 以下の s)

```
1 for s in shapes:
2     s.to_svg()
```

- 継承 (クラスの定義を部分的に再利用可能)
  - 既存のクラスを修正・拡張して新しいクラスを作る
  - 例: circle から, svg_circle を作る

```
1 class svg_circle(circle):
2   def to_svg(self, ...): ...
```

Features of object-oriented languages (review)

- Polymorphism (the same code can take multiple types of data at runtime)
  - a single variable or an expression can be different types at runtime (e.g. `s` below)

```
1 for s in shapes:
2     s.to_svg()
```

- Inheritance (a class can reuse another classes)
  - extend an existing class to create another
  - ex. derive `svg_circle` from `circle`

```
1 class svg_circle(circle):
2   def to_svg(self, ...): ...
```

今日の理解目標

- 多相性も継承も，静的な型付けを諦めれば大した問題ではない (Python)
- 多相性・継承と静的な型付けを両立させようとしている言語
  - C++, Java, Eiffel, OCaml, ...
- だが正しく設計するのは思ったより難しい
- 最終的にどんな言語を作るかは設計者の選択だが,「やってはいけないこと (安全でなくする規則)」は共通．それを理解する.

Objectives of this chapter

- neither polymorphism nor inheritance cause any problem without static type checks (Python)
- many languages have both polymorphism/inheritance and static type checks
  - C++, Java, Eiffel, OCaml, ...
- *it's not easy to design them correctly*
- design details are up to the designer's taste, but there are *"things you ought not to (designs that make it unsafe)."* the goal is to understand them.

Contents

静的型システム (復習)

- 基本的目標: 実行時に型エラーを起こさないこと (型安全性) を，実行前に保証する
- 中心的課題: メソッドの呼び出し式 ($E.m(\ldots)$) に対し, E にメソッド m を持たないようなデータが紛れ込まないことを, 実行前に保証する
- 例: 以下の `s` に, `to_svg` メソッドを持たないようなデータが紛れ込まないことを保証

```
1 for s in shapes:
2     s.to_svg()
```

- 上では「メソッド呼び出し」が失敗しないことに限定して述べているが, フィールドアクセスや組み込みの演算子 (`[]`, `+`) などでも問題はほとんど同じ

Static type systems (review)

- the basic goal: guarantee, *prior to execution*, that type errors do not happen (*type safety*) *during execution*
- central issue: for a method call expression ($E.m(\ldots)$), guarantee, *prior to execution*, that all data that become the value of E have method m
- ex: guarantee `s` below never has data that does not have `to_svg`

```
1 for s in shapes:
2     s.to_svg()
```

- note: we've been only concerned about "method calls"; there are similar issues around field accesses or builtin types, of course, but issues are similar

静的型システム (復習)

- 基本アプローチ:
  - 各式や変数に「静的な型」という属性を持たせ，ある規則で実行前に計算する
  - 式や文の合法性を静的な型を用いて定める
- データが流れるところ (広義の代入) で, 静的な型が検査される
  - 関数適用: $f(a)$
  - 代入: $x := a$
  - データ更新 $x\texttt{->}f\ \texttt{:=}\ a$

Static type systems (review)

- basics:
  - give each expression or variable an attribute called "static type" and calculate it prior to execution
  - determine the validity of expressions and statements based on static types
- static types are checked wherever data flow (assignments in a broad sense)
  - function application: $f(a)$
  - assignment: x `:=` a
  - data update: x`->`f `:=` a

単純な (多相的でない) 静的型システムの安全性

- 「多相性」をあきらめれば, 安全性を保証するのは簡単
- 「多相性」をあきらめる = すべての変数や式が, 「実行時に取りうる型は一種類だけ」と制限する
- そうすれば, 静的な型 ≈ 動的な型
  - 代入 $x{:=}a$ は, x と a の静的な型が, ぴったり一致している時のみ許す
  - 関数適用なども同様の条件でのみ許す (関数の引数の静的な型 = 渡される引数の静的な型)
- 例えば以下は, 静的な型エラーとして拒絶することになる (`s` や `shapes` にどのような型をつければよいのか?)

```
1 shapes.append(svg_rect(...))
2 shapes.append(svg_circle(...))
3 for s in shapes:
4     s.to_svg()
```

Type safety of simple (monomorphic) static type systems

- forget about polymorphism → type safety is easy to achieve
- forget about polymorphism = each variable or expression can have values of only a single type at runtime
- thus static type ≈ dynamic type
  - assignment x `:=` a is allowed only when x and a have an identical static type
  - function applications are allowed in a similar condition (the static type of an input parameter = the static type of the argument expression)
- the following will be rejected (what could the static types of `s` or `shapes` be?)

```
1 shapes.append(svg_rect(...))
2 shapes.append(svg_circle(...))
3 for s in shapes:
4     s.to_svg()
```

多相性と静的型システム

- 目標:
  - 多相性 (= あるひとつの式や変数が「実行時に」複数の型を取り得ること) を許す
  - 実行時の型エラーは起こさない (起きうるプログラムは静的に拒絶する)
- 以下は許したい

```
1 shapes.append(svg_rect(...))
2 shapes.append(svg_circle(...))
3 for s in shapes:
4     s.to_svg()
```

- 以下は許さない (文字列は to_svg メソッドを持たない)

```
1 shapes.append(svg_rect(...))
2 shapes.append(circle(...))
3 for s in shapes:
4     s.to_svg()
```

Polymorphism and static type systems

- goals:
  - polymorphism (= a single expression or a variable can take, at runtime, values of different types)
  - guarantee a (type correct or statically typed) program never raises a type error at runtime (reject programs that could)
- wish to allow this:

```
1 shapes = [ svg_rect(...), svg_circle(...) ]
2 for s in shapes:
3     s.to_svg()
```

- should not allow this (strings do not have `to_svg` method)

```
1 shapes = [ svg_rect(...), svg_circle(...) ]
2 for s in shapes:
3     s.to_svg()
```

静的な (実行前の) 型と動的な (実行時の) 型

- 似てはいるが別物であるという意識が重要
- 意図としては，ある「式」の静的な型 ≈ 実行時にその式の評価結果となり得るデータすべての型の「和」
- 例:

```
1 shapes.append(svg_rect(...))
2 shapes.append(svg_circle(...))
3 for s in shapes:
4     s.to_svg()
```

  - s (を評価した結果) の「動的な型」: circle だったり rect だったり (でも一時にはそのどちらから)
  - s (という式) の「静的な型」: 型システムの設計次第. 例えば
    - 「svg_shape」という (概念的な) 型?
    - 「svg_circle または svg_rect」という型?
    - 「to_svg というメソッドを持つオブジェクト」という型?
    - など (設計次第)

Static (compile-time) types and dynamic (runtime) types

- it's important to understand they are similar but different
- a static type of an expression is meant to be ≈ the "union" of all (runtime) types of values that the expression could take at runtime
- ex:

```
1 shapes.append(svg_rect(...))
2 shapes.append(svg_circle(...))
3 for s in shapes:
4     s.to_svg()
```

  - the runtime type of `s` at a particular evaluation of it is either `circle` or `rect` (not both at the same time)
  - the static type of expression `s` is, for example,
    - ★ an abstract type that might be called `svg_shape`?
    - ★ "`circle` or `rect`"?
    - ★ "object having `to_svg` method"?
    - ★ etc. (details depending on the type system you design)

中心的な問い

- 「ある (静的な) 型 ← 別の (静的な) 型」という広義の代入 (データの流れ) をどのような場合に許すのか?

`rect`	:=	`svg_rect` ?
`svg_shape`	:=	`svg_rect` ?
`svg_shape` のリスト	:=	`svg_rect` のリスト ?
`svg_shape` の配列	:=	`svg_rect` の配列 ?
関数 `svg_shape` → `float`	:=	関数 `svg_rect` → `float`
関数 `int` → `svg_shape`	:=	関数 `int` → `svg_rect`

C++は? Java は?

The central question

- when do we allow an assignment: "a (static) type ← another (static) type"?

`rect`	:=	`svg_rect` ?
`svg_shape`	:=	`svg_rect` ?
list of `svg_shape`	:=	list of `svg_rect` ?
array of `svg_shape`	:=	array of `svg_rect` ?
function `svg_shape` → `float`	:=	function `svg_rect` → `floa`
function `int` → `svg_shape`	:=	function `int` → `svg_rect`

C++? Java?

重要概念：部分型 (subtype)

- 多相性を許す前提で, $x{:=}a$ が合法なのはどのようなときか?
- x の静的な型が T, a の静的な型が T' として, 以下のようなときなのだろう
  - T に対して許される操作はすべて，T' に対しても許される
  - 標語的に言えば「T' が T の一種とみなせる」
- このような T, T' の関係を「T' が T の部分型 (subtype)」であるといい，$T' \leq T$ と書く
- 以降の中心的な問いは,

  $T' \leq T$ と言ってよいのはどういう時か?

- これが確立されれば,

  $x{:=}a$ が合法 $\iff$ a の静的な型 $\leq x$ の静的な型

Important concept : subtype

- when is it OK to allow $x{:=}a$, in the presence of polymorphism?
- it will be, given x's static type is T, and a's static type T', when:
  - all operations applicable to T are also applicable to T'
  - in more intuitive terms: "T' can be seen as T"
- such a relationship between T and T' is called T' is a subtype of T and written $T' \leq T$
- the central question in the rest of today is:

  between which types have there is a relation $T' \leq T$*?*

- with this established,

  $x{:=}a$ *is allowed* $\iff$ *a's static type* $\leq$ *x's static type*

オブジェクト指向言語の静的型システム: 改めて目標

- 最も基本的な目標は, 「型安全であること」だが, それ以外の目標もある
- 目標 1 (言語設計上の目標): 継承による再利用を柔軟に行いたい
  - メソッド，フィールドを「追加する」以外にも，メソッドやフィールドの型を「変更」したくなるときがある
- 目標 2 (実装上の目標): 静的な型を利用して効率的な実行がしたい
  - ≈ フィールド参照，メソッド探索が「ポインタ＋定数オフセットの dereference」くらいですむ

Static type systems of object-oriented languages: additional goals

- the most basic goal is being "type safe," but there are others too
- goal 1 (a design goal): allow inheritance to derive classes flexibly
  - in addition to "adding" methods and fields in a child class, there are occasions when we want to "modify" types of existing methods or fields
- goal 2 (an implementation goal): we wish to execute the program efficiently utilizing static types
  - $\approx$ compile field reference and method search into $\approx$ "memory access with a pointer + a constant offset"

目標 1: 継承による再利用を柔軟に

- 継承 ≈ 既存のクラスを変更して新しいクラスを作る機構
- 変更方法:
  1. フィールド，メソッドを追加
     - ★ 事実上すべての言語でサポートされている
  2. フィールド，メソッドを再定義 (上書き)
     - ★ フィールドやメソッドの型 (返り値，引数) の変更を許すか?
     - ★ そもそもなぜそうしたくなるかを理解するのが先

Goal 1: flexible inheritance

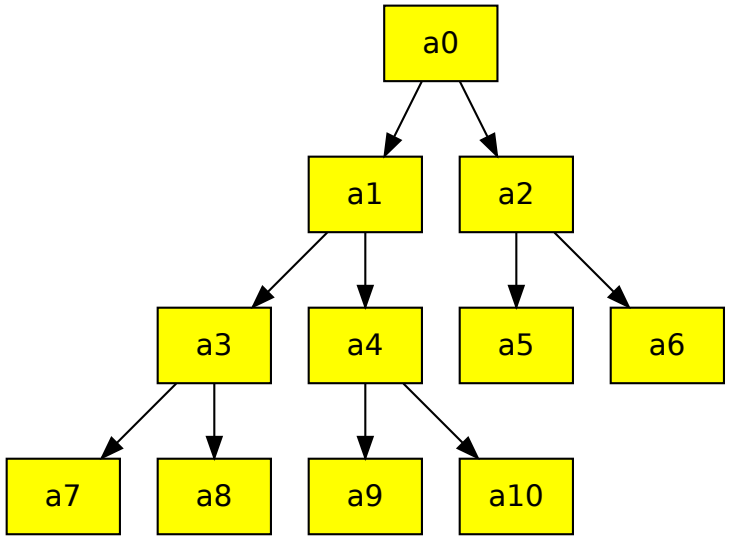
- inheritance $\approx$ a mechanism to create a class by modifying existing classes
- possible ways to modify classes:
  1. add fields or methods
     - ⋆ virtually all languages support this
  2. redefine (overwrite) fields or methods
     - ⋆ should we allow fields or methods (parameters and return values) to change their types in a child class?
     - ⋆ perhaps we should first discuss why we ever want to to this

型の変更が必要・有用な例

2 分木 → 拡張された 2 分木

- 普通の 2 分木:

```
1  struct node {
2    int val;
3    node * left; node * right;
4    node * get_left();
5    void set_left(node *);   };
```

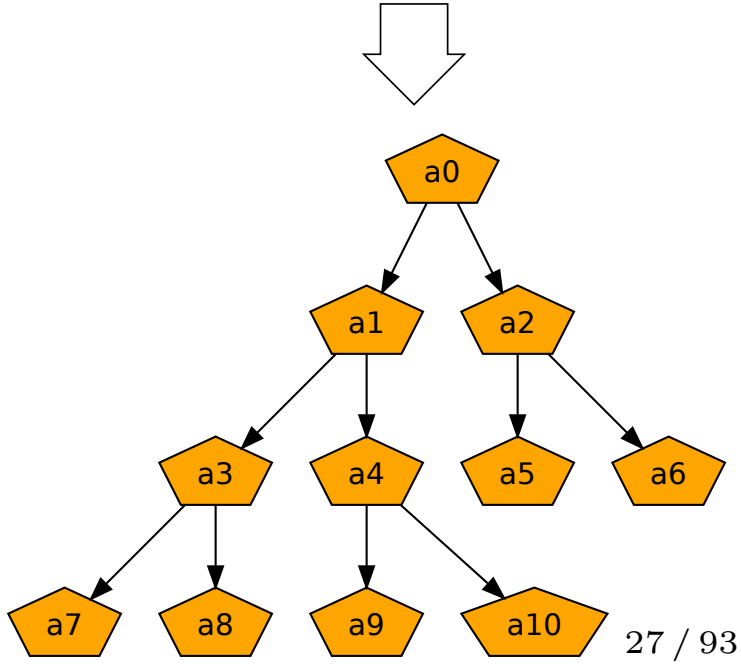


拡張 2 分木 (例: depth を追加)

```
1  struct ex_node :  node {
2    int depth; };
```

≈

```
1  struct ex_node {
2    int val;
3    node * left; node * right;
4    node * get_left();
5    void set_left(node *);
6    int depth; };
```

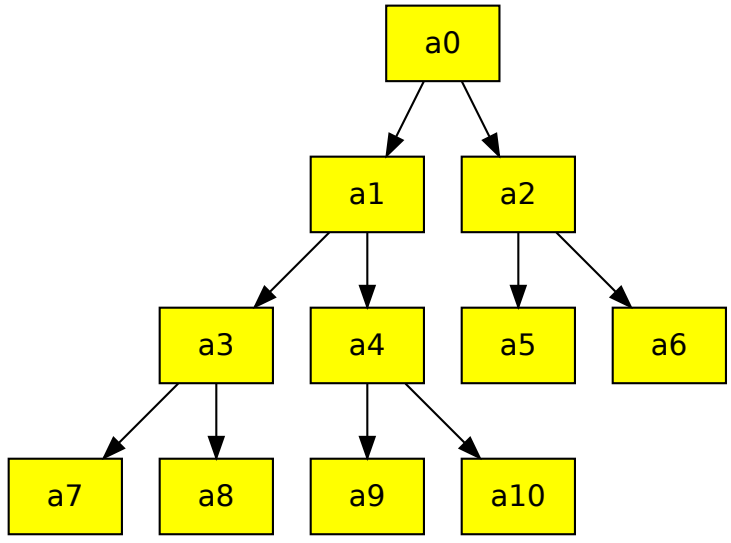


Wish to modify types when inheriting a class

binary tree→extended binary tree

- an ordinary binary tree:

```
1  struct node {
2    int val;
3    node * left; node * right;
4    node * get_left();
5    void set_left(node *);   };
```

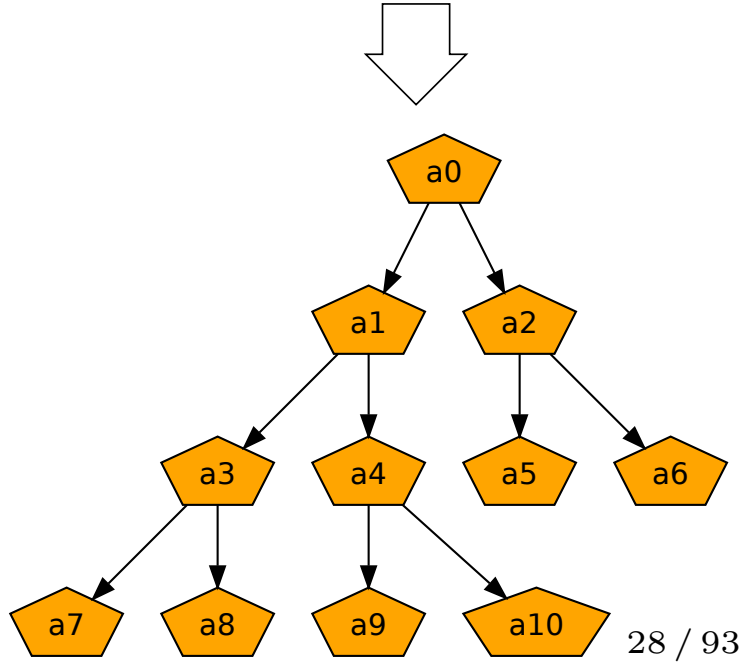


an extended tree (e.g., add `depth`)

```
1  struct ex_node :  node {
2    int depth; };
```

≈

```
1  struct ex_node {
2    int val;
3    node * left; node * right;
4    node * get_left();
5    void set_left(node *);
6    int depth; };
```



2分木の例の考察(1)

- やりたかったこと: node の木を ex_node の木に拡張
- node → node という参照 (left, right など) は，ex_node → ex_node という参照に変更が必要
- それにひきずられて変更が必要なメソッドの引数や返り値もある

```
1  struct ex_node {
2    int val;
3    node * ex_node * left; node * ex_node * right;
4    node * ex_node * get_left();
5    void set_left(node * ex_node *);
6    int depth; };
```

Analyzing the extended tree example (1)

- goal: extend tree of `node` to tree of `ex_node`
- references `node` → `node` (left, right, etc.) need to become references `ex_node` → `ex_node`
- some method arguments and return types need to be accordingly modified too

```
1  struct ex_node {
2    int val;
3    node * ex_node * left; node * ex_node * right;
4    node * ex_node * get_left();
5    void set_left(node * ex_node *);
6    int depth; };
```

2分木の例の考察(2)

- 言い換えれば,

```
1 struct node {
2   int val;
3   node * left; node * right;
4   node * get_left();
5   void set_left(node *);
6 };
```

  の，node *は,「node *およびそれを拡張したもの」というよりは,「自分自身と同じクラス」という方が意図に近い
- 同様の問題が生ずる他の例を考えてみよ

Analyzing the extended tree example (2)

- in other words, `node *` in:

```
1  struct node {
2    int val;
3    node * left; node * right;
4    node * get_left();
5    void set_left(node *);
6  };
```

  intends to be "the class being defined itself," rather than "`node *` and its derived types"

- can you find other examples?

目標2: フィールド参照やメソッド探索の効率的な実装

- 例:

```
1 for a in A:
2     ... a.x + a.y + a.z ...
```

- 「実行時に型エラーが起きない」ことを保証するだけなら,「A に挿入されるオブジェクトはフィールド x，y，z を持つ (+ それらが足し算できる)」ことを保証するだけで良い
- だがそれだけでは効率的な実装は困難

Goal 2: efficient implementation of field reference and method search

- e.g.,

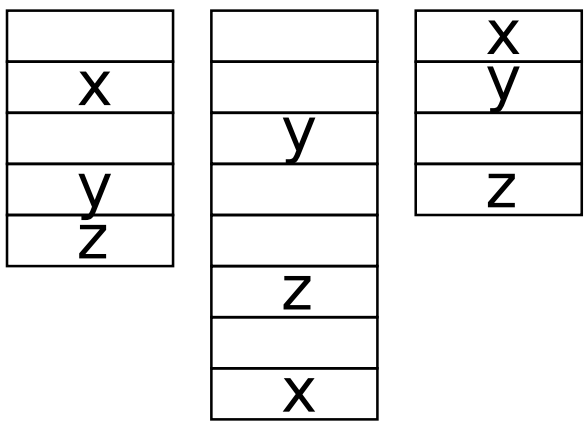
```
1 for a in A:
2     ... a.x + a.y + a.z ...
```

- guaranteeing type safety merely requires `A` to hold only objects having fields `x, y, z` (and their values can be added with +)
- but it's not sufficient for efficient implementation

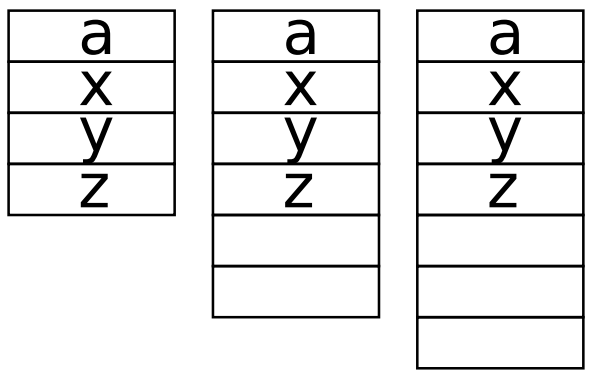
フィールド参照やメソッド探索の効率的な実装

- 効率的な実装 ≈ フィールド参照がポインタ a + 定数オフセットの dereference
- a に代入するものは, 単に「x，y，z を持つ」という以上に, レイアウトが「共通」していてほしい
- 「静的な型を持つ」オブジェクト指向言語の多くはこれを念頭に,「部分型関係 (ひいては合法な関数適用・代入)」の規則を決める

効率的実装 難



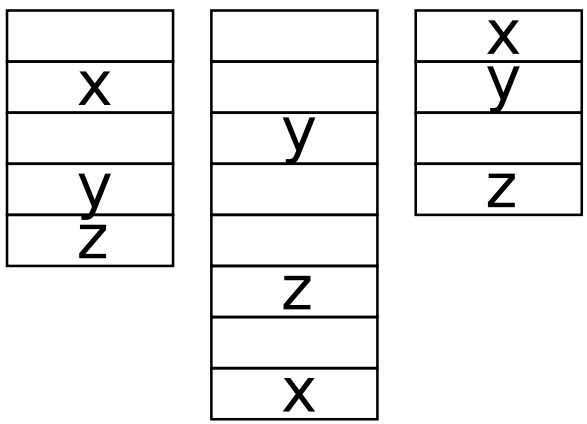
効率的実装 易



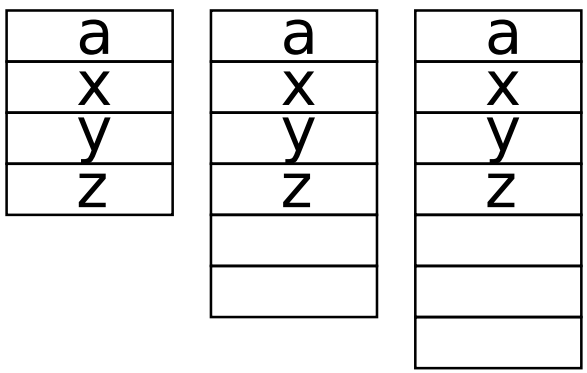
Efficient implementation of field reference and method search

- efficient implementation ≈ *field reference is compiled into dereferencing a pointer with constant offset*
- i.e., all objects assigned to `a` should not only have fields `x`, `y`, `z` but also share "a common layout"
- many statically typed object-oriented languages have this in mind when determining subtype relations (thus legitimacy of function applications and assignments)

効率的実装 難



効率的実装 易



Contents

静的な型を持つオブジェクト指向言語に多く見られる基本方針

- $T' \leq T \approx T'$ が T またはそれを継承したもの
  - 標語的には「継承 $\approx$ 部分型」という方針
  - それが必然の方針ではないことを注意しておく
- また, これは T や T' がクラスであった場合で, 部分型は全ての型に定義する必要がある
  - 配列 (T' の配列 $\leq T$ の配列?)
  - リスト (T' のリスト $\leq T$ のリスト?)
  - 関数 ($T' \to \texttt{int} \leq T \to \texttt{int}$, $\texttt{int} \to T' \leq \texttt{int} \to T$?)
  - ...
- それらの取り扱いは言語ごとに異なる

A common approach to designing subtypes in statically typed object-oriented languages

- $T' \leq T \approx T'$ is T or a subclass of it
  - in rough terms, "inheritance $\approx$ subtype"
  - note that it is *not* necessary
- also note that this rule only determines the subtype relation among classes; it must be defined among *all* types
  - arrays (array of $T' \leq$ array of T?)
  - lists (list of $T' \leq$ list of T?)
  - functions ($T' \to \texttt{int} \leq T \to \texttt{int}$, $\texttt{int} \to T' \leq \texttt{int} \to T$?)
  - ...
- different languages have different rules

ここまでのまとめ

- 実装上の観点から，多くの言語で，「$T' \leq T$ であるのは T' が T かそれを継承したクラス」の場合 (だけ) としている
- 設計上の観点から，継承元 (親) クラスの型を継承先 (子) クラスで変更したいことがある
- 要詳細検討 (言語ごとに異なる)
  - 部分型関係の規則 ($\rightarrow$ 合法な代入や関数適用の規則) を，配列，関数その他，クラス以外の型へどう拡張するか
  - 継承時にフィールド，メソッドの型の変更を許すか
- 実例を通して，落とし穴を理解する

Summary, so far

- many languages, due to its implementation advantages, have a rule along the line of: $T' \leq T \iff T'$ is T or a child class of it
- a child class wants to *modify types* that appeared in the parent class (the class you inherit from)
- things every language must specify
  - how to define subtype relations ($\rightarrow$ which assignments or function applications are allowed) for types other than classes
  - which modification to fields or methods are allowed upon inheritance
- we'll see design pitfalls through examples

Contents

具体例：Java

- 部分型の規則:
  - クラス: T' が T を継承 (extends) していれば $T' \leq T$

```
1 class B extends A { ... };
2 A a = new B(); // 合法
```

  - 配列: $T' \leq T \rightarrow T'$ の配列 $\leq T$ の配列

```
1 A[] a = new B[10]; // 合法
```

  - 配列以外の総称型: $T' \leq T \rightarrow C\texttt{<}T'\texttt{>} \leq C\texttt{<}T\texttt{>}$ではない

```
1 ArrayList<A> a = new ArrayList<B>(); // NG
```

  - 一貫性なし．どちらが「理にかなっている」のか?
- 継承時の型の変更
  - フィールド: 不可
  - メソッドの返り値: 可
  - メソッドの引数: 不可 (複数の，異なるメソッドになる．動的に選ばれる訳ではない)

A case study : Java

- subtype rule:
  - among classes: T' extends $T \rightarrow T' \leq T$

```
1 class B extends A { ... };
2 A a = new B(); // allowed
```

  - among arrays: $T' \leq T \rightarrow$ array of $T' \leq$ array of T

```
1 A[] a = new B[10]; // allowed
```

  - among generic types (other than arrays): $T' \leq T \not\rightarrow C\texttt{<}T'\texttt{>} \leq C\texttt{<}T\texttt{>}$

```
1 ArrayList<A> a = new ArrayList<B>(); // NG
```

  - inconsistent. which one makes more sense?
- type modifications upon inheritance
  - of fields: not allowed
  - of method return values: not allowed
  - of method parameters: not allowed (new methods have distinct methods, not subject to dynamic bindings)

T の配列 $:= T'\ (\leq T)$ の配列

- B が A を継承しているとする
- Java では,「A の配列 := B の配列」が静的型検査を通る
- だが以下を実行すると型エラーになる
- 6 行目で ArrayStoreException

```
1 static void check_array() {
2   A a = new A();      // A = A
3   B b = new B();      // B = B
4   B[] ab = new B[1]; // B[] = B[]
5   A[] aa = ab;        // A[] = B[]
6   aa[0] = a;          // A = A (実行時:  B = A)
7   B bb = ab[0];       // B = B (実行時:  B = A)
8   System.out.println(bb.y);
9 }
```

array of T := array T' $(\leq T)$

- suppose `B` extends `A`
- in Java, an assignment "array of `A` := array of `B`" passes static type checks (by the compiler)
- it still results in type errors *when executed*!
- `ArrayStoreException` happens at line 6

```
1  static void check_array() {
2    A a = new A();      // A = A
3    B b = new B();      // B = B
4    B[] ab = new B[1]; // B[] = B[]
5    A[] aa = ab;        // A[] = B[]
6    aa[0] = a;          // A = A (at runtime:  B = A)
7    B bb = ab[0];       // B = B (at runtime:  B = A)
8    System.out.println(bb.y);
9  }
```

Java の ArrayStoreException

- 配列への代入 $a[i] = x$ において，x の「実行時の」型が $a[i]$ の実行時の型またはその部分型でない時に発生
- 本当は，それが起きないことを保証するのが静的な型システムの目標だったのだが，そうはなっていなかった
- 実行時にそれを検出するという苦肉の策

`ArrayStoreException` in Java

- it happens upon an array update $a[i] = x$, when x's actual type (the type revealed at runtime) is not $a[i]$'s actual type or a subtype of it
- (presumably) the static type system is *meant* to catch this (prior to execution, but it did not
- it resorted to detect it at runtime

そもそもどこで間違ったのだろう?

- B ≤ A のとき，B[] ≤ A[] は「一見」問題がなさそう
  - 「長方形」は「図形」の一種 ⇒ 「長方形の配列」は「図形の配列」の一種
  - この直感 (型 ≈ 集合) だけではやや大雑把すぎた
- もう少し精密に「A[] に許される操作はすべて，B[] に許されるか?」と考える
  - 要素参照：問題無さそう
  - 要素代入：
    - ★ A[] には A の要素を代入できる
    - ★ B[] には A の要素を代入できない
- もし要素の代入がなかったら? そのとおり， B[] ≤ A[] は正しい

Where did we go wrong?

- the rule: `B` $\leq$ `A` $\rightarrow$ `B[]` $\leq$ `A[]` "looks" reasonable
  - "rectangle" is a kind of "shape" $\Rightarrow$ "array of rectangles" is a kind of "array of shapes"
  - this intuitive reasoning (a type $\approx$ a set) wasn't accurate enough and needs to take a closer look
- to be more accurate, ask: "is any operation legitimate on `A[]` also legitimate on `B[]`?"
  - element reference `A[i]`: looks OK
  - element update :
    - ⋆ `A[]` can update its element with an object of `A`
    - ⋆ `B[]` cannot update element with an object of `A`
- what if they don't have support update? yes, `B[]` $\leq$ `A[]` *is* safe

配列だけの問題ではないはず

- リスト，ハッシュ表，木構造など色々な「コンテナ」で同じ問題が生ずるはず
- 例：ArrayList<T>
- パターンは全く同じだがこちらは「正しく」静的型エラー

```
1  static void check_list() {
2    A a = new A();
3    B b = new B();
4    ArrayList<B> lb = new ArrayList<B>();
5    ArrayList<A> la = lb; // 静的型エラー
6    la.add(a);
7    B bb = lb.get(0);
8    System.out.println(bb.y);
9  }
```

It should not be only about arrays

- other "containers" (lists, hash tables, trees, etc.) should have similar issues
- ex : `ArrayList<T>`
- you can make exactly the same pattern, which is "correctly" rejected

```
1  static void check_list() {
2    A a = new A();
3    B b = new B();
4    ArrayList<B> lb = new ArrayList<B>();
5    ArrayList<A> la = lb; // a compile-time error
6    la.add(a);
7    B bb = lb.get(0);
8    System.out.println(bb.y);
9  }
```

どうしてこうなった!?

http://wiki.c2.com/?JavaArraysBreakTypeSafety

- リスト，ハッシュ表，木構造など色々な「コンテナ」で同じ問題が生ずるはず
- 例：`ArrayList<T>`
- パターンは全く同じだがこちらは「正しく」静的型エラー

```
1  static void check_list() {
2    A a = new A();
3    B b = new B();
4    ArrayList<B> lb = new ArrayList<B>();
5    ArrayList<A> la = lb; // 静的型エラー
6    la.add(a);
7    B bb = lb.get(0);
8    System.out.println(bb.y);
9  }
```

It should not be only about arrays

- other “containers” (lists, hash tables, trees, etc.) should have similar issues
- ex : `ArrayList<T>`
- you can make exactly the same pattern, which is “correctly” rejected

```
1  static void check_list() {
2    A a = new A();
3    B b = new B();
4    ArrayList<B> lb = new ArrayList<B>();
5    ArrayList<A> la = lb; // a compile-time error
6    la.add(a);
7    B bb = lb.get(0);
8    System.out.println(bb.y);
9  }
```

継承時の型変更

事実上不可能

- 子供で同じ名前のフィールド，メソッドを定義したら，上書き (変更) になるわけではない
- フィールド：名前が同じでも別物 (混乱のもと)
- メソッド：引数の個数・型が違うものは別のメソッド
  - どれが呼ばれるかは, 呼び出し時の引数の個数と静的な型で決まる
- 引数の個数・型が同じ場合のみ上書き

```
1  class A {
2    A x;
3    A get() { ... }
4    void set(A a) { ... }
5  };
6  class B extends A {
7    B x;                    // 上のx とは「別の」フィールド
8    B get() { ... }         // A::get()を上書き
9    void set(B b)  ...     // set(A a)とは「別の」メソッド
10 };
```

Java on changing types upon inheritance

practically, not allowed at all

- if a child class defines a field or a method of the same name, it won't overwrite (change) the one in the parent
- field : different fields under the same name (source of confusion, at best)
- method : different methods if arity or parameter types are different
  - which one gets called is determined by the number of arguments and their static types of a method call expression
- overwritten only when the arity and parameter types match

```
1  class A {
2    A x;
3    A get() { ... }
4    void set(A a) { ... }
5  };
6  class B extends A {
7    B x;                    // another x, different from x in A
8    B get() { ... }         // overwrites A::get()
9    void set(B b)  ...    // another method, different from A::set(A a)
```

2 分木の例

- 2 分木のノードを継承して「拡張 2 分木のノード」を作っても,「拡張されたノードだけ」の木構造は保証されない

```
1 class node {
2     node left;
3     node right;  };
4 class ex_node extends node {
5     int depth;   };
```

- 無理やり再定義してもそれは別のフィールドであって, 領域のムダ, 混乱のもと

```
1 class ex_node extends node {
2     ex_node left;  // node left とは別のフィールド
3     ex_node right; // node right とは別のフィールド
4     int depth;   };
```

- 一から作り直す方がマシ
- 「2 分木」「拡張 2 分木」の両方に適用可能なコードを書くのは困難

Binary search tree example

- extending the node class to the "extended node" won't make "a tree of (only) extended nodes"

```
1 class node {
2     node left;
3     node right;  };
4 class ex_node extends node {
5     int depth;   };
```

- even if you add them, they are different fields wasting space and at best the source of confusion

```
1 class ex_node extends node {
2     ex_node left;  // another field distinct from node left
3     ex_node right; // another field distinct from node right
4     int depth;   };
```

- you'd better make it from scratch
- it's difficult to write a code that works both on binary trees and extended binary trees

具体例：C++

- Java の A 型 $\approx$ C++の A* 型
- Java のメソッド $\approx$ C++の `virtual` 宣言されたメソッド

と読みかえれば，C++と Java は似ているが，違いもある

- 部分型の規則
  - クラスへのポインタの部分型: T' が T を継承していれば，$T'* \leq T*$

```
1 class B : public A { ... };
2 A * a = new B(); // 合法
```

  - 配列: $T' \leq T \rightarrow T'* \leq T*$ ではない

```
1 A ** a = new B*[10]; // NG
```

  - 総称型もどき (テンプレート): $T' \leq T \rightarrow C<T'>* \leq C<T>*$ ではない

```
1 vector<A*> * a = new vector<B*>(10); // NG
```

- 継承時の型の変更: Java と似ている (詳細省略)

A case study: C++

- class type A in Java $\approx$ `A*` in C++
- method in Java $\approx$ `virtual` method in C++

With this translation, C++ is similar to but still different from Java

- subtyping rule
  - subtyping between pointer-to-class types: T' extends $T \Rightarrow T'* \leq T*$

```
1 class B : public A { ... };
2 A * a = new B(); // allowed
```

  - arrays: $T' \leq T \not\Rightarrow T'* \leq T*$

```
1 A ** a = new B*[10]; // NG
```

  - pseudo generic types or, templates
    $T' \leq T \not\Rightarrow C<T'>* \leq C<T>*$

```
1 vector<A*> * a = new vector<B*>(10); // NG
```

- rules about changes to types upon inheritance: similar to

Eiffel (1)

- フィールド，メソッドの型の変更を許す機構 (redefine) を持つ
- フィールド，メソッドの引数，返り値のどれも，親クラスに現れた型をその部分型へ書き換えることを許す
- 同じ名前のメソッド，フィールドは常にひとつしかない (当たり前のようだが C++, Java はそうではない)
- Eiffel での 2 分木ノード定義

```
1 class NODE
2 feature
3   left, right : NODE
4   ...
5 end
```

Eiffel (1)

- has a mechanism (redefine) that allows changes to field/method types
- allows field/parameter/return types to *change into their subtypes*
- allows only a single field or method of the same name (seems natural but is not the case in C++ and Java)
- binary tree node in Eiffel

```
1 class NODE
2 feature
3   left, right : NODE
4   ...
5 end
```

Eiffel (2)

- 継承で 2 分木ノードを拡張する

```
1  class EX_NODE
2  inherit NODE
3    redefine left, right, ... end
4  feature
5    left, right : EX_NODE
6    depth : INTEGER
7     ...
8  end
```

- もしくは最初から「自分と同じ型への参照」と書ける

```
1  class NODE
2  feature
3    left, right : like Current
4    ...
5  end
```

- 将来の変更に自動的に追従できる

Eiffel (2)

- extend the binary tree node by inheritance

```
1  class EX_NODE
2  inherit NODE
3    redefine left, right, ... end
4  feature
5    left, right : EX_NODE
6    depth : INTEGER
7     ...
8  end
```

- or you can write from the beginning that a field has “the same type with itself”

```
1  class NODE
2  feature
3    left, right : like Current
4    ...
5  end
```

- the actual type automatically changes when extended

Eiffel (3)

- 部分型の規則:
  - クラス: T' が T を継承したクラスならば，$T' \leq T$
  - 配列: $T' \leq T$ ならば，$ARRAY[T'] \leq ARRAY[T]$
  - 総称型: $T' \leq T$, C が総称型のとき，$C[T'] \leq C[T]$
- 一見美しいルールだが,
  - 2 は Java 同様，間違っている (実行時の型エラーがあり得る)
  - 3 も間違っている (配列のような型を総称型を用いて作れる)
  - 1 も，型の再定義があるおかげで間違っている (次のスライド)

Eiffel (3)

- subtyping rules
  - classes: T' inherits $T \rightarrow T' \leq T$
  - arrays: $T' \leq T \rightarrow ARRAY[T'] \leq ARRAY[T]$
  - generics: $T' \leq T$ and C is a generic type $\rightarrow C[T'] \leq C[T]$
- they look beautiful, but
  - #2 is making the same mistake as Java (it may cause runtime type errors)
  - #3 is also wrong (we can make an array-like type using generic types)
  - #1 is also wrong, due to redefinition of types upon inheritance (next slide)

型の再定義が壊す安全性

- あるクラスのフィールドやメソッドの型を，継承時に無制限に変更したら,「子クラス $\leq$ 親クラス」になるとは限らないのは自明

```
1 class A { int x; };
```

を

```
1 class B extends A { String x; };
```

にしてはまずいのは自明

- そこで Eiffel は，ある型を「部分型へ」変更することのみを許しているが，それでも「子クラス $\leq$ 親クラス」とはならない場合がある

Type safety violated by type redefinition

- if you allow types to be redefined *arbitrarily* in a child class, then, obviously, "child class $\leq$ parent class" won't hold. Extending

```
1 class A { int x; };
```

into

```
1 class B extends A { String x; };
```

won't make `B` $\leq$ `A`, obviously.

- Eiffel thus only allows a type to be *redefined to its subtype*, but there are still cases where "child class $\leq$ parent class" won't hold

型の再定義が壊す安全性

- 馴染みやすさのため Java 風の文法で説明
- $B \leq A$ とする
- 親クラス RefA:

```
1  class RefA {
2    A x; // フィールド
3    A get() { } // 返り値
4    void set(A y) { } // 引数
5  };
```

- 子クラス RefB (A を B に変更)

```
1  class RefB : RefA {
2    redefine x, get, set; // 再定義 (できると仮定する)
3    B x; // フィールド
4    B get() { } // 返り値
5    void set(B y) { } // 引数
6    void get2() {  } // 追加
7  };
```

- RefB の再定義は全て, A を B (A の子クラス) にしたもの
- 本当に, RefB を RefA の部分型とみなして良いか?

Type safety broken by type redefinition

- explain with Java-like syntax (for your familiarity)
- assume $B \leq A$
- define a parent class `RefA`:

```
1  class RefA {
2    A x; // field
3    A get() { } // return value
4    void set(A y) { } // input parameter
5  };
```

- its child class `RefB` (change $A \to B$)

```
1  class RefB : RefA {
2    redefine x, get, set; // redefinition (assume it possible)
3    B x; // field
4    B get() { } // return value
5    void set(B y) { } // input parameter
6    void get2() {  } // a new method
7  };
```

- type redefinitions in `RefB` all change `A` to `B` (a child class of `A`)
- Q: *can we say* `RefB` *is a subtype of* `RefA` *???*

実行時型エラーを起こす例

```
1 A a = new A();
2 RefB bb = new RefB();
3 RefA aa = bb;          // RefB = RefA
4 aa.set(a);             // RefB::set(B y)にA を渡している(B = A)
```

This program causes a runtime error

```
1  A a = new A();
2  RefB bb = new RefB();
3  RefA aa = bb;          // RefB = RefA
4  aa.set(a);             // passing A to RefB::set(B y) (i.e., B = A)
```

正しい考え方

RefB を RefA の部分型とみなして良いか? $\iff$ RefA に行える操作はすべて RefB に行えるか?

- RefA に行える操作:
  - `get()`
  - `set(A a)`
  - `x` の参照
  - `x` への `A` の代入
- RefB に行ってよいか?
  - `get()` :
  - `set(A a)` :
  - `x` の参照 :
  - `x` への `A` の代入 :

正しい考え方

RefB を RefA の部分型とみなして良いか? $\Longleftrightarrow$ RefA に行える操作はすべて RefB に行えるか?

- RefA に行える操作:
  - `get()`
  - `set(A a)`
  - `x` の参照
  - `x` への `A` の代入
- RefB に行ってよいか?
  - `get()` : OK
  - `set(A a)` :
  - `x` の参照 :
  - `x` への `A` の代入 :

正しい考え方

RefB を RefA の部分型とみなして良いか? $\Longleftrightarrow$ RefA に行える操作はすべて RefB に行えるか?

- RefA に行える操作:
  - `get()`
  - `set(A a)`
  - `x` の参照
  - `x` への `A` の代入
- RefB に行ってよいか?
  - `get()` : OK
  - `set(A a)` : NG (`B` の `set` は `B` しか受け取れない)
  - `x` の参照 :
  - `x` への `A` の代入 :

正しい考え方

RefB を RefA の部分型とみなして良いか? $\Longleftrightarrow$ RefA に行える操作はすべて RefB に行えるか?

- RefA に行える操作:
  - `get()`
  - `set(A a)`
  - `x` の参照
  - `x` への `A` の代入
- RefB に行ってよいか?
  - `get()` : OK
  - `set(A a)` : NG (`B` の `set` は `B` しか受け取れない)
  - `x` の参照 : OK
  - `x` への `A` の代入 :

正しい考え方

RefB を RefA の部分型とみなして良いか? $\Longleftrightarrow$ RefA に行える操作はすべて RefB に行えるか?

- RefA に行える操作:
  - `get()`
  - `set(A a)`
  - `x` の参照
  - `x` への `A` の代入
- RefB に行ってよいか?
  - `get()` : OK
  - `set(A a)` : NG (`B` の `set` は `B` しか受け取れない)
  - `x` の参照 : OK
  - `x` への `A` の代入 : NG (`B` の `x` は `B` しか代入できない)

The correct thinking

Is `RefB` *a subtype of* `RefA`*?* $\iff$ *are all operations applicable to* `RefA` *also applicable to* `RefB`*?*

- applicable to `RefA`:
  - `get()`
  - `set(A a)`
  - reference `x`
  - set `A` to `x`
- are they applicable to `RefB`?
  - `get()` :
  - `set(A a)` :
  - reference `x` :
  - set `A` to `x` :

The correct thinking

Is `RefB` *a subtype of* `RefA`*?* $\iff$ *are all operations applicable to* `RefA` *also applicable to* `RefB`*?*

- applicable to `RefA`:
  - `get()`
  - `set(A a)`
  - reference `x`
  - set `A` to `x`
- are they applicable to `RefB`?
  - `get()` : OK
  - `set(A a)` :
  - reference `x` :
  - set `A` to `x` :

The correct thinking

Is `RefB` *a subtype of* `RefA`*?* $\Longleftrightarrow$ *are all operations applicable to* `RefA` *also applicable to* `RefB`*?*

- applicable to `RefA`:
  - `get()`
  - `set(A a)`
  - reference `x`
  - set `A` to `x`
- are they applicable to `RefB`?
  - `get()` : OK
  - `set(A a)` : NG (`set` of `B` only take `B` (or its subtype))
  - reference `x` :
  - set `A` to `x` :

The correct thinking

Is `RefB` *a subtype of* `RefA`*?* $\Longleftrightarrow$ *are all operations applicable to* `RefA` *also applicable to* `RefB`*?*

- applicable to `RefA`:
  - `get()`
  - `set(A a)`
  - reference `x`
  - set `A` to `x`
- are they applicable to `RefB`?
  - `get()` : OK
  - `set(A a)` : NG (`set` of `B` only take `B` (or its subtype))
  - reference `x` : OK
  - set `A` to `x` :

The correct thinking

Is `RefB` *a subtype of* `RefA`*?* $\iff$ *are all operations applicable to* `RefA` *also applicable to* `RefB`*?*

- applicable to `RefA`:
  - `get()`
  - `set(A a)`
  - reference `x`
  - set `A` to `x`
- are they applicable to `RefB`?
  - `get()` : OK
  - `set(A a)` : NG (`set` of `B` only take `B` (or its subtype))
  - reference `x` : OK
  - set `A` to `x` : NG (`x` in `B` only takes `B` (or its subtype))

Contents

もう何も信じられない?

- 「継承」してできたものは「部分型」というのは思いこみ
- ましてやクラス以外の型には継承などない (それでも「部分型」の概念はある)
- 何を「部分型」とみなしてよいのか「正しく」理解することが先
- 基本はいつも同じ:

$$T' \leq T \iff T \text{ に適用可能な操作がすべて } T' \text{に適用可能}$$

Became a paranoid?

- "inheritance is not subtyping"
- it's simply *wrong* to think a child class (made by inheritance) is necessarily a subtype of the parent
- non-class types do not have inheritance (but still have subtype relationships)
- it is important to firmly understand *when a type is a subtype of another*
- the principle is always this:

$$T' \leq T \iff \text{any operation applicable to} T \text{is applicable to} T'$$

問題の要約 (1) 間違った部分型

1. 配列または mutable な要素に関する間違い

$$(*) \quad \alpha' \leq \alpha \Rightarrow \alpha'\text{の配列} \leq \alpha\text{の配列}$$

```
1  α[] aa = new α'[1];   // α の配列 <- α' の配列
2  aa[0] = new α;        // α <- α
```

2. 引数の narrowing に関する間違い (Eiffel)

$$(*) \quad \Rightarrow \quad \begin{array}{l} \alpha' \leq \alpha \\ \texttt{class A'}\,\{\,\texttt{void take}(\alpha' : x)\,\{\ldots\}\} \\ \leq \texttt{class A}\,\{\,\texttt{void take}(\alpha : x)\,\{\ldots\}\} \end{array}$$

```
1  A a = new A';   // A <- A'
2  a.take(new A);  // A <- A
```

3. 上記を一歩遡ると以下が勘違い

$$(*) \quad \alpha' \leq \alpha \quad \Rightarrow \quad \alpha' \to \beta \leq \alpha \to \beta$$

Summary of issues (1) wrong subtypes

1. errors around arrays or mutable fields

$$(*) \qquad \alpha' \leq \alpha \Rightarrow \alpha'\text{の配列} \leq \alpha\text{の配列}$$

```
1  α[] aa = new α'[1];   // α の配列 <- α' の配列
2  aa[0] = new α;        // α <- α
```

2. errors around narrowing of arguments (Eiffel)

$$(*) \qquad \begin{array}{l} \alpha' \leq \alpha \\ \Rightarrow \quad \texttt{class A'}\,\{\,\texttt{void take}(\alpha' : x)\,\{\ldots\}\} \\ \quad\; \leq \texttt{class A}\,\{\,\texttt{void take}(\alpha : x)\,\{\ldots\}\} \end{array}$$

```
1  A a = new A';   // A <- A'
2  a.take(new A);  // A <- A
```

3. the root of the above errors is the following misunderstanding:

$$(*) \qquad \alpha' \leq \alpha \;\Rightarrow\; \alpha' \to \beta \leq \alpha \to \beta$$

問題の要約 (2) 継承時の柔軟性の無さ

- 継承時にメソッドやフィールドの型を再定義したくなる
- 最も典型的な場面: 再帰的なデータ構造
  - リスト, 木, グラフ, ...
  - 例: ノードに「重み」を追加
  - 次ノード, 子ノード, 隣接ノード, etc. の型を変更したくなる

```
1 class list_node { list_node next; }
2 class tree_node { tree_node left, right; }
3 class graph_node { graph_node[] neighbors; }
```

- 多くの言語では単純に, 「出来ない」
- 「継承 $\rightarrow$ 部分型」にこだわる言語は, ここでも間違え安い

Summary of issues (2) inflexible inheritance

- we wish to redefine method or field types upon inheritance
- the most typical situation: recursive data types
  - lists, trees, graphs, ...
  - ex: add a "weight" to nodes
  - wish to change types of next node, child nodes, neighbor nodes, etc.

```
1 class list_node { list_node next; }
2 class tree_node { tree_node left, right; }
3 class graph_node { graph_node[] neighbors; }
```

- many languages simply don't allow this
- languages sticking to the misconception: "inheritance → subtyping" tend to make a flawed type system here too

部分型を基本から理解する

- **関数の部分型:** ここを正しく理解することが話の 80%

$$s' \to t' \leq s \to t \iff s' \geq s \text{ and } t' \leq t$$

- **レコードの部分型:** 記法

$$\{a_0 : s_0,\ a_1 : s_1,\ \ldots\}$$

で, フィールド a_0 を持ちその型が s_0, フィールド a_1 を持ちその型が s_1, ... という構造体 (オブジェクト, レコードなど) を表すとすると,

$$\begin{aligned} & \{b_0 : t_0, b_1 : t_1, \ldots\} \leq \{a_0 : s_0, a_1 : s_1, \ldots\} \\ \iff\ & \{b_0, b_1, \ldots\} \supset \{a_0, a_1, \ldots\} \\ \wedge\ & \forall i, j\ a_i = b_j \to t_j \leq s_i \\ \wedge\ & \forall i, j\ a_i = b_j \quad \text{and } a_i\ (b_j) \text{ is mutable} \to t_j = s_i \end{aligned}$$

Understanding subtypes from the fundamentals

- **subtypes between function types:** if you understand this, you are 80% done

$$s' \to t' \leq s \to t \iff s' \geq s \text{ and } t' \leq t$$

- **subtypes between record types:** let

$$\{a_0 : s_0,\ a_1 : s_1,\ \ldots\}$$

represent a structural type (a class or a record) having field a_0 of type s_0, a_1 of type s_1, and so on.
then,

$$\begin{array}{rl} & \{b_0 : t_0, b_1 : t_1, \ldots\} \leq \{a_0 : s_0, a_1 : s_1, \ldots\} \\ \iff & \{b_0, b_1, \ldots\} \supset \{a_0, a_1, \ldots\} \\ \wedge & \forall i, j\ a_i = b_j \to t_j \leq s_i \\ \wedge & \forall i, j\ a_i = b_j \text{and } a_i\ (b_j) \text{ is mutable} \to t_j = s_i \end{array}$$

共変 (covariant) と反変 (contravariant)

- 型 α を要素に含む T があるとする (e.g., 関数型は引数の型, 返り値の型を含む; 配列型は要素の型を含む, ...)
- 以下の問いにパッと答えられるようになることが重要:
  α をその部分型 α' に置き換えたときにできる型 T'
  $(= T\{\alpha \mapsto \alpha'\})$ は T の部分型か?
- 定義: T が α に関して

  共変 (covariant): $\alpha' \leq \alpha \to T\{\alpha \mapsto \alpha'\} \leq T$

  反変 (contravariant): $\alpha' \leq \alpha \to T \leq T\{\alpha \mapsto \alpha'\}$

  不変 (invariant): 共変でも反変でもない

Covariant and contravariant

- say type T has another type α as a component (e.g., a function type has an input type and an output type; an array type has its element type, ...)
- it's quite useful if we are able to answer the following question quickly:
  if we replace α with its subtype α', is the resulting type T' $(= T\{\alpha \mapsto \alpha'\})$ a subtype of T?
- definition: T is, with respect to α,

covariant :	$\alpha' \leq \alpha \to T\{\alpha \mapsto \alpha'\} \leq T$
contravariant :	$\alpha' \leq \alpha \to T \leq T\{\alpha \mapsto \alpha'\}$
invariant :	neither covariant nor contravariant

知見の要約

言葉で言えば,

1. 関数の型 $(\alpha \to \beta)$ は,
   - その返り値の型 (β) に関して coavariant,
   - その引数 (α) に関して contravariant
2. レコードの型は,
   - immutable なフィールドの型に関しては covariant,
   - mutable なフィールドの型に関しては invariant

実はこの二つさえ理解すればほぼ全てが理解できる:

- オブジェクト $\approx$ 関数 (メソッド) をフィールドに持つレコード
- 配列 $\approx$ 要素の参照 (get)，変更 (set) をメソッドに持つオブジェクト

Summary of the insights

in short terms:

1. a function type $(\alpha \to \beta)$ is
   - coavariant in its output type (β)
   - contravariant in its input type (α)
2. record type is
   - covariant in immutable fields
   - invariant in mutable fields

everything follows from the two principles

- objects$\approx$ records having functions (methods) as fields
- arrays$\approx$ objects having `get` and `set` methods (or records whose fields are all mutable)

しつこいけど...

- 改めて

$$\begin{aligned} & \alpha' \leq \alpha \\ \Rightarrow\ & \texttt{class A}' \texttt{ \{ void take}(\alpha' : x) \texttt{ \{...\}\}} \\ & \leq \texttt{class A \{ void take}(\alpha : x) \texttt{ \{...\}\}} \end{aligned}$$

  が成り立つか問うてみる
- オブジェクトはレコードのようなもの

$$\texttt{A} \approx \{\texttt{take} : \alpha \to \texttt{void}\},\ \texttt{A}' \approx \{\texttt{take} : \alpha' \to \texttt{void}\}$$

- 前スライドの規則を思い出すと,

$$\begin{aligned} \texttt{A}' \leq \texttt{A} \ \Rightarrow\ & \{\texttt{take} : \alpha' \to \texttt{void}\} \leq \{\texttt{take} : \alpha \to \texttt{void}\} \\ \Rightarrow\ & \alpha' \to \texttt{void} \leq \alpha \to \texttt{void} \\ \Rightarrow\ & \alpha \leq \alpha' \end{aligned}$$

To reiterate . . .

- let's ask again whether the following holds

$$\begin{array}{rl} & \alpha' \leq \alpha \\ \Rightarrow & \texttt{class A'}\,\{\,\texttt{void take}(\alpha' : x)\,\{\ldots\}\} \\ & \leq \texttt{class A}\,\{\,\texttt{void take}(\alpha : x)\,\{\ldots\}\} \end{array}$$

- objects are like records

$$\texttt{A} \approx \{\texttt{take} : \alpha \to \texttt{void}\},\ \texttt{A'} \approx \{\texttt{take} : \alpha' \to \texttt{void}\}$$

- recall the rules in the last slide

$$\begin{array}{rcl} \texttt{A'} \leq \texttt{A} & \Rightarrow & \{\texttt{take} : \alpha' \to \texttt{void}\} \leq \{\texttt{take} : \alpha \to \texttt{void}\} \\ & \Rightarrow & \alpha' \to \texttt{void} \leq \alpha \to \texttt{void} \\ & \Rightarrow & \alpha \leq \alpha' \end{array}$$

配列

- 改めて

$$\begin{aligned} & \alpha' \leq \alpha \\ \Rightarrow \quad & \alpha'\text{の配列} \leq \alpha\text{の配列} \end{aligned}$$

  が成り立つか問うてみる
- α の配列は以下のレコードのようなもの

$$\{\texttt{get} : \texttt{int} \to \alpha,\ \texttt{put} : \texttt{int} \to \alpha \to \texttt{void}\}$$

- put の型を見て直ちに違うとわかる．丁寧に書くと，

$$\begin{aligned} & \alpha'\text{の配列} \leq \alpha\text{の配列} \\ \Rightarrow \quad & \{\texttt{get} : \texttt{int} \to \alpha',\ \texttt{put} : \texttt{int} \to \alpha' \to \texttt{void}\} \\ & \leq \{\texttt{get} : \texttt{int} \to \alpha,\ \texttt{put} : \texttt{int} \to \alpha \to \texttt{void}\} \\ \Rightarrow \quad & \texttt{int} \to \alpha' \to \texttt{void} \leq \texttt{int} \to \alpha \to \texttt{void} \\ \Rightarrow \quad & \alpha' \to \texttt{void} \leq \alpha \to \texttt{void} \\ \Rightarrow \quad & \alpha \leq \alpha' \end{aligned}$$

Arrays

- let's ask again the following holds

$$\begin{array}{rl} & \alpha' \leq \alpha \\ \Rightarrow & \text{array of}\alpha' \leq \text{array of}\alpha \end{array}$$

- array of α is like the following record type

$$\{\texttt{get} : \texttt{int} \to \alpha,\ \texttt{put} : \texttt{int} \to \alpha \to \texttt{void}\}$$

- we'll immediately see it does not hold by looking at the type of `put`. in fact,

$$\begin{array}{rl} & \text{array of}\alpha' \leq \text{array of}\alpha \\ \Rightarrow & \{\texttt{get} : \texttt{int} \to \alpha',\ \texttt{put} : \texttt{int} \to \alpha' \to \texttt{void}\} \\ & \leq \{\texttt{get} : \texttt{int} \to \alpha,\ \texttt{put} : \texttt{int} \to \alpha \to \texttt{void}\} \\ \Rightarrow & \texttt{int} \to \alpha' \to \texttt{void} \leq \texttt{int} \to \alpha \to \texttt{void} \\ \Rightarrow & \alpha' \to \texttt{void} \leq \alpha \to \texttt{void} \\ \Rightarrow & \alpha \leq \alpha' \end{array}$$

まとめ (おつかれ様です)

- オブジェクト指向 $\approx$ 部分型 ($\leq$) による多相と継承
- $T' \leq T \sim T$ に行える操作は全部 T' にも行える
- T を継承して出来たクラス T' が, T の部分型になるとは限らない (継承 $\neq$ 部分型)
- $T' \leq T$ でも, $C[T'] \leq C[T]$ とは限らない
  - $T' \to \beta \not\leq T \to \beta$
  - $T \to \beta \leq T' \to \beta$
  - array of $T' \not\leq$ array of T
- 実際に上記を間違えて設計された言語がある (Java, Eiffel)
- 他の言語の仕様を見てみて

Summary (finally!)

- object-oriented $\approx$ subtype ($\leq$) polymorphism and inheritance
- allow x := a when a's type $\leq$ x's type
- $T' \leq T \sim$ operations applicable to T are all applicable to T' too
- class T' derived from T is not necessarily a subtype of T (inheritance $\neq$ subtyping)
- $T' \leq T$ does not imply $C[T'] \leq C[T]$
  - $T' \to \beta \not\leq T \to \beta$
  - $T \to \beta \leq T' \to \beta$
  - array of $T' \not\leq$ array of T
- real languages fell into these traps (Java, Eiffel, etc.)
- take a look at the spec of other languages