TAPL 輪講

24 Existential Types

2010/6/16

秋山 茂樹

24 Existential Types

1. Motivation
2. Data Abstraction with Existentials
   - Abstract Data Types
   - Existential Objects
   - Objects vs. ADTs
3. Encoding Existentials

24.1
Motivation

∀X.T の二つの解釈

- logical intuition
  - 型∀X.T の要素はすべてのSについて
    [X→S]T なる型を持つ値である
- operational intuition
  - 型∀X.T の要素は型Sを引数に取り型[X→S]T
    の項を返す関数である (type abstraction)

∃X.T の二つの解釈

- logical intuition
  - 型∃X.T の要素はあるSについて

    [X→S]T なる型を持つ値である

- operational intuition
  - 型∃X.T の要素は型Sと型[X→S]Tの項のペアである (∃X.T は {∃X,T} と書く)

Existential Types とは

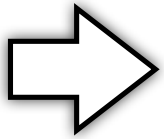
- 特定の型を隠蔽するための型 ∃X. T
  - パッケージ、モジュールを表現するのに使用
  - 例: ∃X. {a:X, f:X→X}
    - 何かよくわからない型Xに対して
      - Xの値a
      - Xの値に対する操作f
    - が定義されている？

Existential Introduction

- 型 S と項 t のペア:　{*S, t} : {∃X, T}
  - S には項 t 内にある隠蔽したい型を指定 (hidden representation type)
  - 例

p = {*Nat, {a=5, f=λx:Nat. succ(x)}}

- p : {∃X, {a:X, f:X→X}}
- p : {∃X, {a:X, f:X→Nat}}



{*S, t} は複数の型を持ち得る

‘as’ による型注釈

- {*S, t} as {∃X, T}

p = {*Nat, {a=5, f=λx:Nat. succ(x)}}
as {∃X, {a:X, f:X→X}}

▸ p : {∃X, {a:X, f:X→X}}

例

p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}}
as {∃X, {a:X, f:X→Nat}}

▸ p4 : {∃X, {a:X, f:X→Nat}}

p5 = {*Bool, {a=true, f=λx:Bool. 0}}
as {∃X, {a:X, f:X→Nat}}

▸ p5 : {∃X, {a:X, f:X→Nat}}

無意味な package

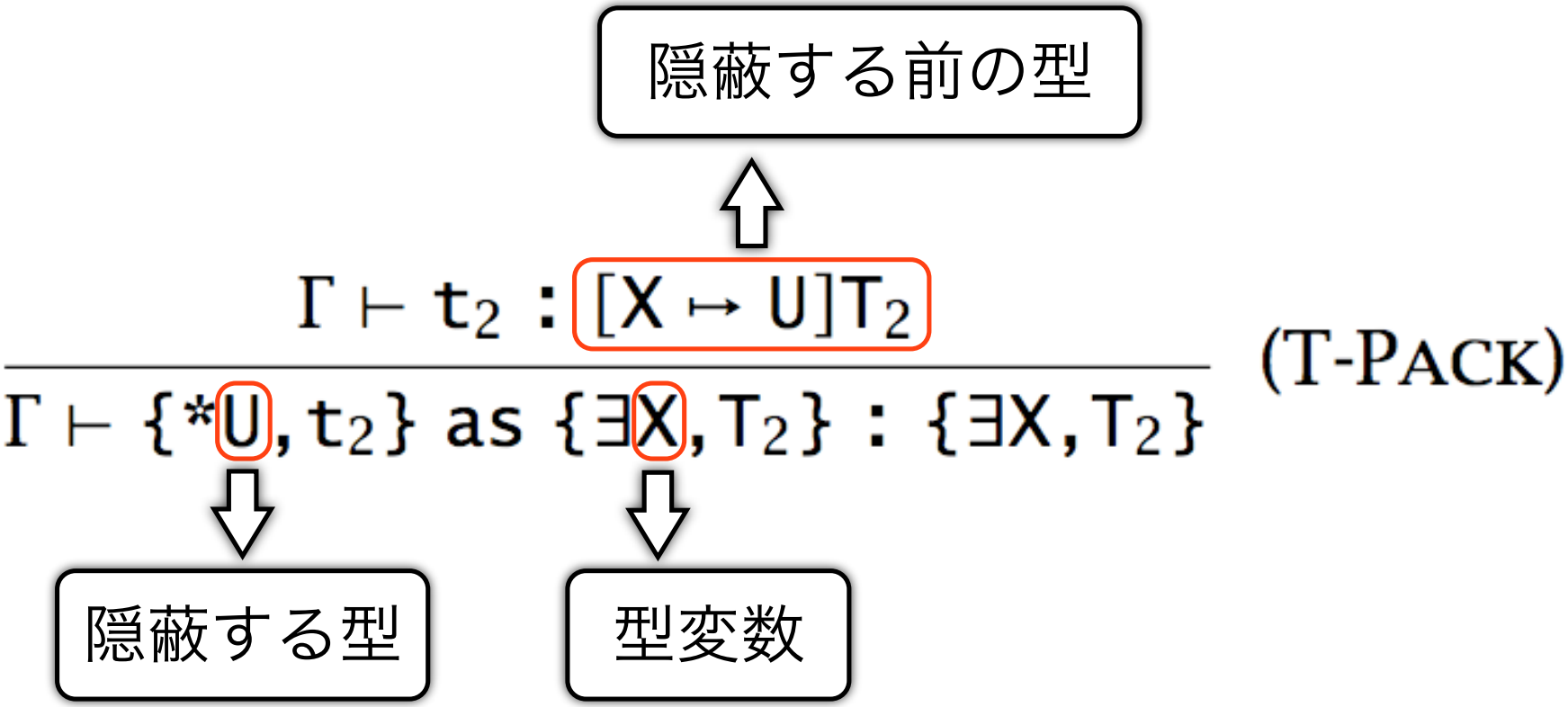
p6 = {*Nat, {a=0, f=λx:Nat. succ(x)}}
as {∃X, {a:X, f:X→X}}

p7 = {*Nat, {a=0, f=λx:Nat. succ(x)}}
as {∃X, {a:X, f:Nat→X}}

p8 = {*Nat, {a=0, f=λx:Nat. succ(x)}}
as {∃X, {a:Nat, f:Nat→Nat}}

T-PACK

- Existential Introduction の型付け規則



Existential Elimination

- module における open に対応
  - 隠蔽された型と値に名前をつける

let {X, x} = p4 in (x.f x.a)
▸ 1 : Nat

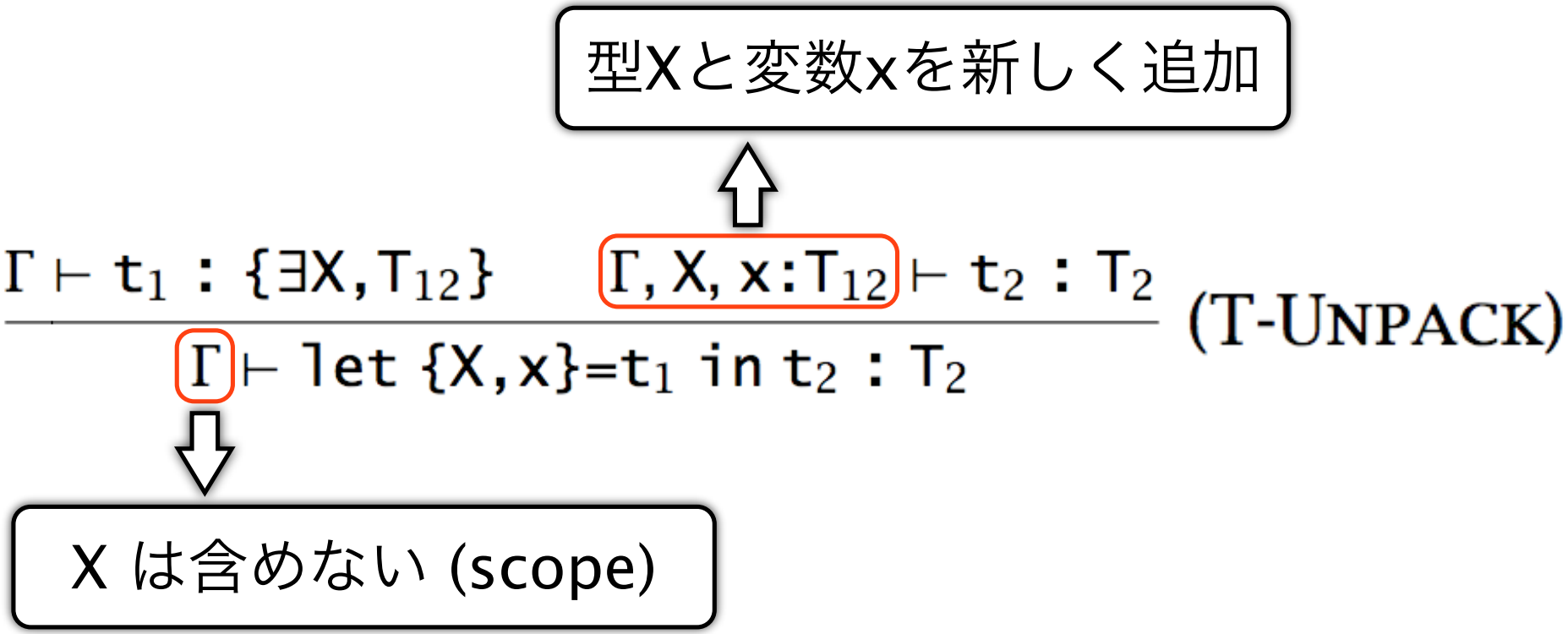
let {X, x} = p4 in succ(x.a)
▸ Error: argument of succ is not a number

let {X, x} = p4 in x.a
▸ Scoping Error!

p4 : {∃X, {a:X, f:X→Nat}}

T-UNPACK

- Existential Elimination の型付け規則



E-UNPACKPACK

value になるまで置き換えてから unpack

$$\texttt{let } \{X, x\} = (\{*T_{11}, v_{12}\} \texttt{ as } T_1) \texttt{ in } t_2 \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2 \quad \text{(E-UnpackPack)}$$

24.2
Data Abstraction with Existentials

前提

- purely functional
- subtyping, inheritance は扱わない

Abstract Data Types

- *abstract data type* (ADT) の構成要素

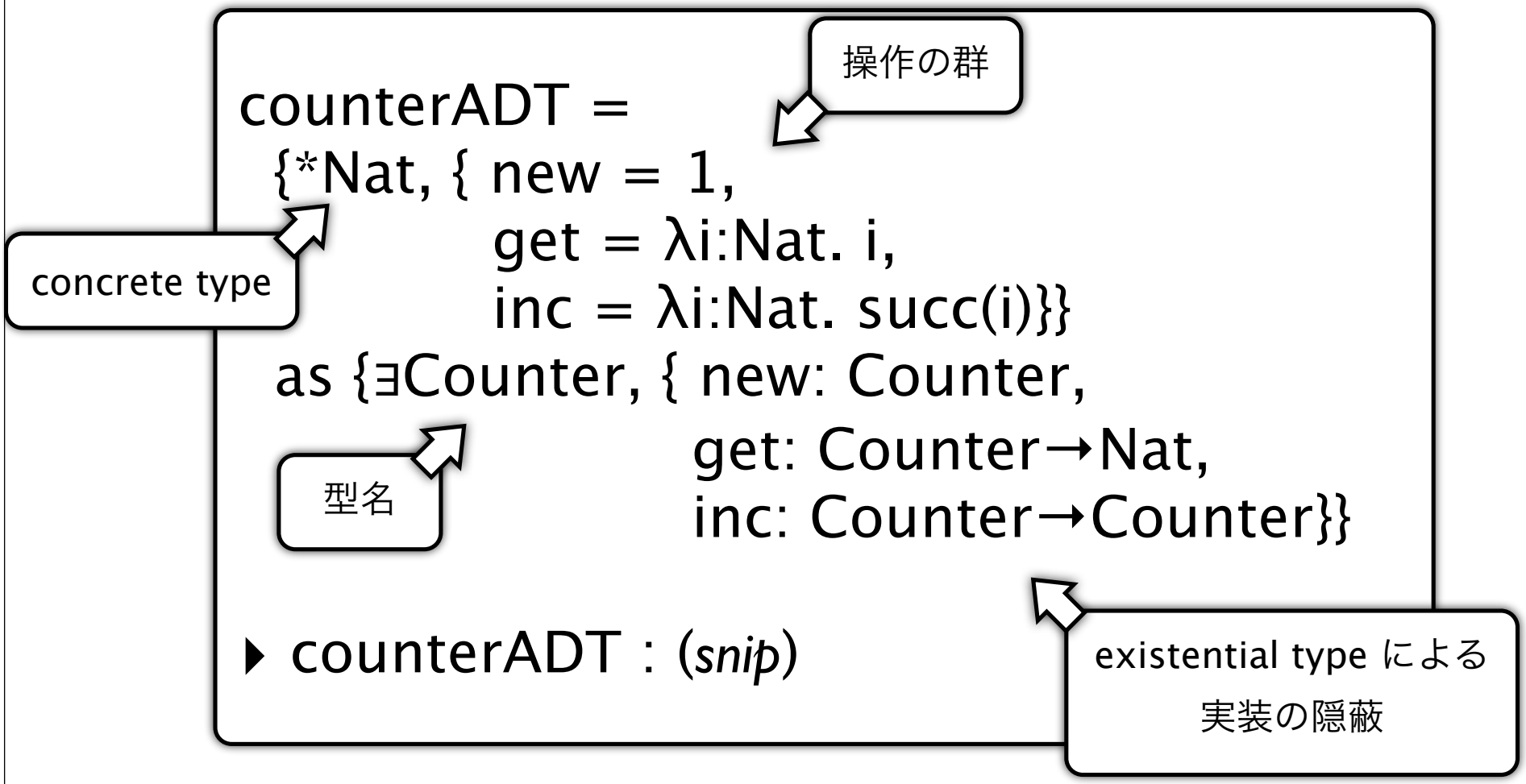
  (1) 型名A

  (2) 具体的な型T (concrete representation type)

  (3) Tの値に対する操作の群 (生成、更新など)

  (4) *abstraction boundary* (実装の隠蔽)

ADT: 例


操作の群
counterADT =
{*Nat, { new = 1,
get = λi:Nat. i,
inc = λi:Nat. succ(i)}}
as {∃Counter, { new: Counter,
get: Counter→Nat,
inc: Counter→Counter}}
concrete type
型名
counterADT : (snip)
existential type による
実装の隠蔽

Existential Objects

- object の構成要素

  (1) 内部状態

  (2) メソッド群

  (3) 内部状態の隠蔽

Object: 例

```
c = {*Nat,
      { state = 5,
        methods = { get = λx:Nat. x,
                    inc = λx:Nat. succ(x)}}}
     as Counter
where:
  Counter = {∃X, {state:X,
                  methods: {get:X→Nat,
                            inc:X→X}}}
```

▸ c : (*snip*)

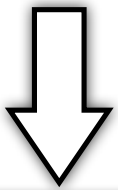
問題

let {X, body} = c in body.methods.get(body.state)

- 5 : Nat

let {X, body} = c in body.methods.inc(body.state)

- Error: Scoping Error!



inc : X→X なのでlet式の型に自由変数が出現してしまう

Object ではなく内部状態が返っている

解決策

- repackage

```
c1 =
  let {X, body} = c in
    {*X,
      { state = body.methods.inc(body.state),
        methods = body.methods }}
    as Counter

▸ c1 : Counter
```

ADTs vs. Objects: Package

- ADT
  - 内部状態の型を隠蔽するのに使用
  - 定義した後すぐに open する
- Object
  - オブジェクトの中身を隠蔽するのに使用
  - 内部状態を変更する時まで open しない

ADTs vs. Objects: 抽象型

- ADT
  - 内部状態の型そのもの
- Object
  - 内部状態の型
  - object に対する操作群（メソッド）
    - メソッドを持つことによる柔軟性

Object の柔軟性

- 多くの異なる実装を同時に使うことが可能
  - 型は同じ
  - 例: Window 型
    - TextWindow: 内部表現は文字列
    - ContainerWindow: 内部表現は Window list

ADTの場合

- 同じ型を持っていても同じものとして扱えない
  - let {TextWindow, tw} = t1 in
  - let {ContainerWindow, cw} = t2 in ...
  - t1 と t2 のシグネチャが同じでも tw と cw は別の型として扱われる
    - 定義の後すぐに open してしまうため
- 同じことをするなら Variant を使う

Object の場合

- 同じ型を持つオブジェクトは同様に扱える
  - let tw = ... in let cw = ... in ...
  - tw と cw の型は Window 型
    - 常に pack されているため
- 継承、サブタイピングを実現可能

二項演算子の扱い

- Object では strong binary operations を扱えない
  - strong binary operations
    - データ型の内部表現を知っていないと定義不可能
  - week binary operations
    - データ型の内部表現を知らずに定義可能
    - こちらは扱える（object の型をrecursiveにする必要あり）

strong binary operations の object による定義

NatSet = {∃X, {state:X, methods:
{ empty:X,
singleton:Nat→X,
member:X→Nat→Bool,
union:X→NatSet→X }}}

pack されているので内部の実装が見えない

ADT vs. Objects: まとめ

- ADT
  - Strong binary operations の定義が可能
- Object
  - メソッドの挙動を変更可能

    (subtyping, inheritance などが可能)

24.3
Encoding Existentials

Encoding Existentials

$$\{\exists X, T\} = \forall Y.\, (\forall X.\, T \to Y) \to Y$$

$$\{*S, t\} \text{ as } \{\exists X, T\} = \lambda Y.\, \lambda f{:}(\forall X.\, T \to Y).\, f\ [S]\ t$$

$$\text{let } \{X, x\} = t_1 \text{ in } t_2 = t_1\ [T_2]\ (\lambda X.\, \lambda x{:}T_{11}.\, t_2)$$

24 Existential Types

1. Motivation
2. Data Abstraction with Existentials
   - Abstract Data Types
   - Existential Objects
   - Objects vs. ADTs
3. Encoding Existentials