Types and Programming Languages

26. Bounded Quantification

秋山 茂樹

TAPL Seminar (2011/2/17)

26. Bounded Quantification

パラメトリック多相とサブタイピングを
組み合わせたシステムである $F_{<:}$ の導入

- 応用例: Java/C# の Generics

```
/* 数列の和を求める */
T <T extends Addable> sumList(List<T> list) {
    T sum = new T(0);
    for (Addable x : list)
      sum = sum.add(x);
    return sum;
}
```

リストの要素は
Addable であれば何でもOK

戻り値の型は Addable
ではなく T

発表の流れ

1. Motivation
2. Definitions
3. Examples
4. Safety
5. Bounded Existential Types

1. Motivation

System F + $\lambda_{<:}$

パラメトリック多相とサブタイピングを
「うまく」組み合わせる

- 愚直な組み合わせは問題なく可能だが...
  - パラメトリック多相とサブタイピングは完全に直交
- 工夫することで、より柔軟なシステムを実現可能
  - 例: `λx.{orig=x, asucc=x.a}` の型付け

1. Motivation

System F + λ<:

例1: 恒等関数 λx.x (ただし x はフィールドaをもつレコード)

- サブタイピングを用いて実装すると...

```
  f = λx:{a:Nat}. x
▸ f : {a:Nat} → {a:Nat}

  f {a=0}
▸ {a=0} : {a:Nat}

  f {a=0, b=true}
▸ {a=0, b=true} : {a:Nat}
```

{a:Nat, b:Bool} で
あってほしい

1. Motivation

System F + λ<:

例1: 恒等関数 λx.x (ただし x はフィールドaをもつレコード)

- パラメトリック多相を用いて実装すると...

```
  fpoly = λX. λx:X. x
▸ f : ∀X. X → X

▸ fpoly [{a:Nat}] {a=0}
▸ {a=0} : {a:Nat}

  fpoly [{a:Nat, b:Bool}] {a=0, b=true}
▸ {a=0, b=true} : {a:Nat, b:Bool}
```

universal types を使えば
{a:Nat, b:Bool} と型付け可能

1. Motivation

System F + λ<:

例2: 関数 λx.{orig=x, asucc=succ(x.a)}

- サブタイピングを用いて実装すると...

```
  f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)}
▸ f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}

  f2 {a=0, b=true}
▸ {orig={a=0, b=true}, assuc=1}
                  : {orig:{a:Nat}, asucc:Nat}
```

{a:Nat, b:Bool} で
あってほしい

1. Motivation

System F + λ<:

例2: 関数 `λx.{orig=x, asucc=succ(x.a)}`

- パラメトリック多相を用いて実装すると...

```
  f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)}
▸ Error: Expected record type
```

xがフィールドaを持っているかどうか
わからないため型付けできない

1. Motivation

Bounded Quantification ($F_{<:}$)

型変数に対しサブタイプ関係を指定可能にする

- bounded quantifier: λx<:T.t

Xは{a:Nat}のサブタイプでないといけない

```
  f2poly = λX<:{a:Nat}. λx:X. {orig=x, asucc=succ(x.a)}
▸ f2poly : ∀X<:{a:Nat}. X → {orig:X, asucc:Nat}
```

xがフィールドaを持っているかどうか
bounded quantifier から判断可能

2. Definitions

2つの $F_{<:}$

サブタイプ関係を2種類定義できる

- **kernel $F_{<:}$**
  - 柔軟性よりもシンプルさを重視
- **full $F_{<:}$**
  - シンプルさよりも柔軟性を重視
  - 「$T_1 <: S_1$ ならば $\forall X<:S_1.\ T <: \forall X<:T_1.\ T$」を許す

2. Definitions

kernel $F_{<:}$ の構文

System F をベースに bounded quant. を追加

項

$$
\begin{aligned}
t ::= \;& x \\
& \lambda x{:}T.t \\
& t\ t \\
& \lambda X{<:}T.t \\
& t\ [T]
\end{aligned}
$$

型

$$
\begin{aligned}
T ::= \;& X \\
& \mathrm{Top} \\
& T \to T \\
& \forall X{<:}T.T
\end{aligned}
$$

値

$$
\begin{aligned}
v ::= \;& \lambda x{:}T.t \\
& \lambda X{<:}T.t
\end{aligned}
$$

型環境

$$
\begin{aligned}
\Gamma ::= \;& \varnothing \\
& \Gamma, x{:}T \\
& \Gamma, X{<:}T
\end{aligned}
$$

2. Definitions

kernel $F_{<:}$ の評価規則

評価には影響なし

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TApp)}$$

構文の追加による変更

$$(\lambda X{<:}T_{11}.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAppTAbs)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

2. Definitions

kernel $F_{<:}$ のサブタイピング規則

環境 および 量化子付き型の関係 を追加

Subtyping $\boxed{\Gamma \vdash S <: T}$

$$\frac{}{\Gamma \vdash S <: S} \quad \text{(S-Refl)}$$

$$\frac{\Gamma \vdash S <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash S <: T} \quad \text{(S-Trans)}$$

$$\frac{}{\Gamma \vdash S <: \mathsf{Top}} \quad \text{(S-Top)}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad \text{(S-TVar)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S-Arrow)}$$

$$\frac{\Gamma, X <: U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: U_1 . S_2 <: \forall X <: U_1 . T_2} \quad \text{(S-All)}$$

環境から
サブタイピング関係を
引っ張ってくる

量化子のついた型
についての
サブタイピング関係

2. Definitions

kernel $F_{<:}$ の型付け規則

型抽象/型適用に bounded quant. を許す

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1 . t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

構文に合わせて
量化子を追加

$$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1 . t_2 : \forall X <: T_1 . T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X <: T_{11} . T_{12} \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

適用する型が
サブタイピング関係を
満たすように修正

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

2. Definitions

full $F_{<:}$ のサブタイピング規則

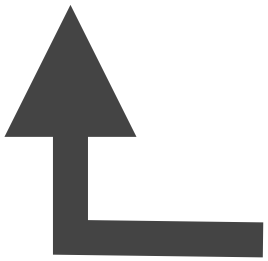
「$T_1 <: S_1$ ならば $\forall X<:S_1.\ T <: \forall X<:T_1.\ T$」を許す

- contravariant subtyping

New subtyping rules

$$\boxed{\Gamma \vdash S <: T}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma, X<: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<: S_1 . S_2 <: \forall X<: T_1 . T_2} \text{(S-ALL)}$$



Subtyping rules (kernel $F_{<:}$)

$$\frac{\Gamma, X<: U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<: U_1 . S_2 <: \forall X<: U_1 . T_2} \text{(S-ALL)}$$

Exercise 26.2.1

Γ = B<:Top, X<:B, Y<:X として

$$\Gamma \vdash B \to Y <: X \to B$$

の subtyping derivation を描け

2. Definitions

Exercise 26.2.2, 26.2.3

full $F_{<:}$ でないと成り立たないサブタイピング関係にある 2 つの型を挙げよ

full $F_{<:}$ でないと成り立たないサブタイピング関係を利用した有用な例を挙げよ

3. Examples

いくつかの例

$F_{<:}$ の有用でシンプルな例を説明する

- Products (略)
  - うまく定義すると次の性質が得られる
    - $S_i <: T_i$ なら Pair S_1 S_2 <: Pair T_1 T_2
- Records (略)
  - うまく定義すると Products と同様の性質が得られる
- Church Encodings

3. Examples

チャーチ数

「第二引数に対して第一引数をn回適用する関数」を自然数 n に対応付ける

- System F による表現

```
CNat = ∀X. (X → X) → X → X
```

- F<: による表現

```
SNat = ∀X<:Top. ∀S<:X ∀Z<:X. (X → S) → Z → X
```

0 の場合は Z,
1以上の場合は S

3. Examples

$F_{<:}$ によるチャーチ数の表現

サブタイプとして零型と正整数型を定義可能

- 自然数型

```
SNat  = ∀X<:Top. ∀S<:X. ∀Z<:X. (X → S) → Z → X
```

- 零型

```
SZero = ∀X<:Top. ∀S<:X. ∀Z<:X. (X → S) → Z → Z
```

- 正整数型

```
SPos  = ∀X<:Top. ∀S<:X. ∀Z<:X. (X → S) → Z → S
```

3. Examples

零型

属する値は 0 としてふるまうもののみ

```
SZero = ∀X<:Top. ∀S<:X. ∀Z<:X. (X → S) → Z → Z

szero = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. z
szero : SZero
```

戻り値型を Z にするためには
z を返さなければならない

3. Examples

正整数型

属する値は正整数としてふるまうもののみ

```
SPos   = ∀X<:Top. ∀S<:X. ∀Z<:X. (X → S) → Z → S

sone   = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s z
sone   : SPos

stwo   = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s z)
stwo   : SPos

sthree = λX. λS<:X. λZ<:X. λs:X→S. λz:Z. s (s (s z))
sthree : SPos
```

戻り値型を S にするためには
z に対して s を一回以上適用した値を返さなければならない

3. Examples

$F_{<:}$ による演算の再定義

「より強い」型を与えることができる

```
ssucc    = λn:SNat.
             λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
               s (n [X] [S] [Z] s z)
▸ ssucc    : SNat → SPos

spluspp = λn:SPos. λm:SPos.
             λX. λS<:X. λZ<:X. λs:X→S. λz:Z.
               (n [X] [S] [S] s (m [X] [S] [Z] s z))
▸ spluspp : SPos → SPos → SPos
```

succ の戻り値は必ず正整数

正整数同士の和は必ず正整数

4. Safety

(kernel) $F_{<:}$ の型安全性

これまでと同様に次の二つの定理を証明する

THEOREM [PRESERVATION]: If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$. □

THEOREM [PROGRESS]: If t is a closed, well-typed $F_{<:}$ term, then either t is a value or else there is some t' with $t \longrightarrow t'$. □

4. Safety

Preservation の証明

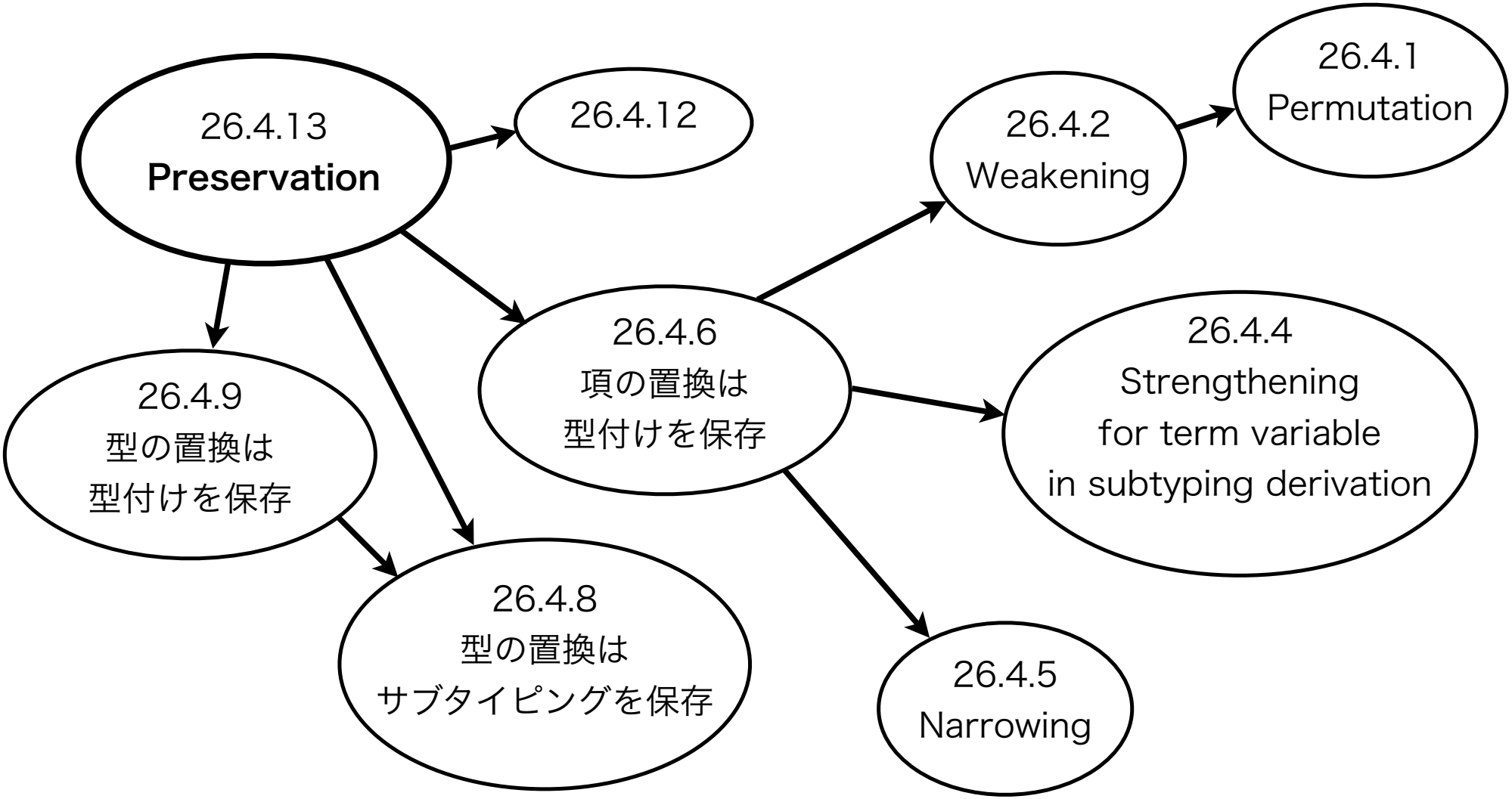
THEOREM [PRESERVATION]: If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$. □

Γ ⊢ t : T の導出に関する帰納法で証明する

- 各型付け規則について
  - t の形から適用可能な評価規則で場合分け
  - t の部分式の型を求め、それを利用して Γ ⊢ t' : T を示す

4. Safety

Preservation の証明に必要な補題



4. Safety

Progress の証明

THEOREM [PROGRESS]: If t is a closed, well-typed $F_{<:}$ term, then either t is a value or else there is some t′ with t ⟶ t′. □

「 ⊢ t : T の導出に関する帰納法で証明する

- 各型付け規則について次を調べる
  - 項 t の形が closed でない
  - 項 t の形が value である
  - 項 t がある t’ について t --> t’ である

評価規則のいずれかを
適用できることを示す

4. Safety

Progress の証明に必要な補題

Canonical Forms Lemma:

- 型 T1 -> T2 をもつ閉じた値は λ x. t の形である
- 型 ∀X<:S1. S2 をもつ閉じた値は λ X<:S1. t の形である

5. Bounded Existential Types

Bounded Existentials

存在型とサブタイピングを組み合わせる

```
  counterADT =
    {*Nat, {new=1, get=λi:Nat.i, inc=λi:Nat.succ(i)}}
    as {∃Counter<:Nat,
          {new:Counter, get:Counter→Nat
                      , inc:Counter→Counter}}
▸ counterADT : (snip)

  let {Counter, counter} = counterADT in
  succ (succ (counter.inc counter.new))
▸ 4 : Nat
```

(counter.inc counter.new) : Counter から
(counter.inc counter.new) : Nat への coercion が発生

カウンタの生成方法のみ隠蔽し、
操作は Nat のものを使用できるようになっている

5. Bounded Existential Types

Definitions

全称型と同じ形で導入

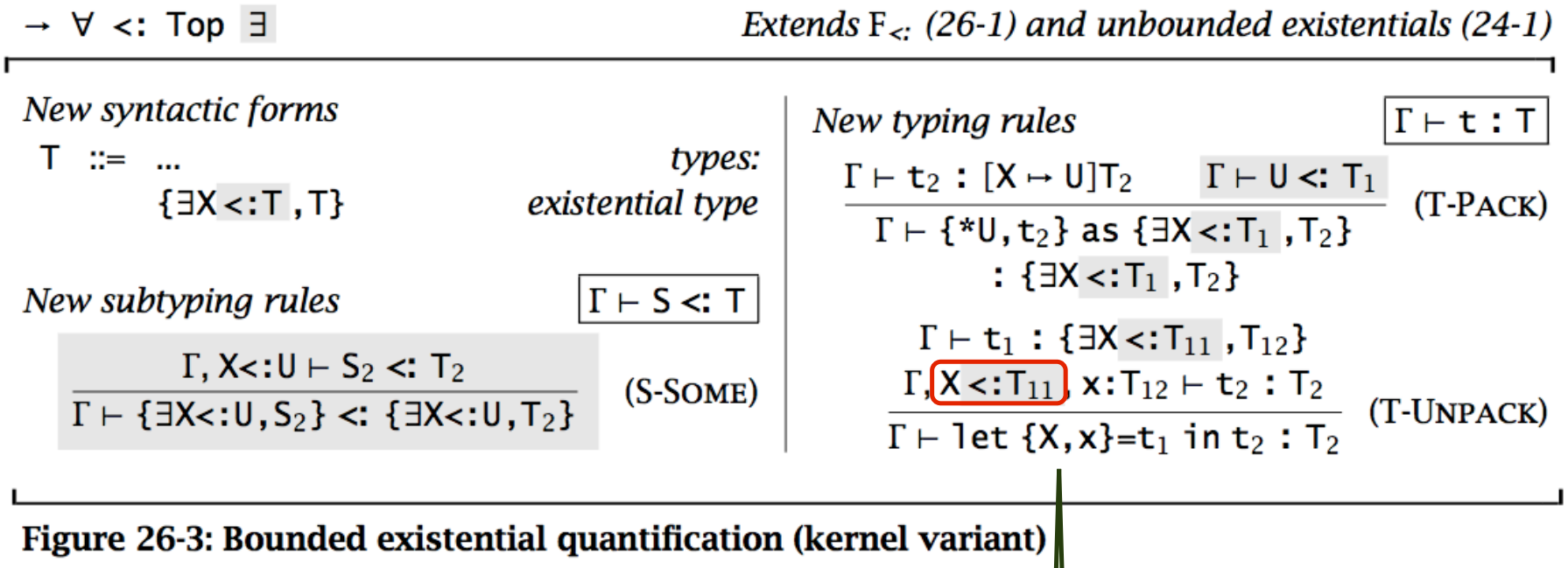


Figure 26-3: Bounded existential quantification (kernel variant)

unpack した後に
指定したサブタイプ関係が環境に追加される

5. Bounded Existential Types

例: object の public field

bounded quant. で公開したいフィールドを指定

```
counter = {*{x:Nat, private:Bool},
  {state = {x = 5, private = false},
   methods = {get = λs:{x:Nat}. s.x,
               inc = λs:{x:Nat, private:Bool}.
                       {x = succ(s.x),
                        private = s.private}}}
  as {∃X<:{x:Nat}, {state:X,
                    methods: {get:X→Nat, inc:X→X}}}

▸ counter : (snip)
```

フィールド x, private のうち x だけを公開する