

Lecture 19 – Existential Types

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Subtyping
 - STLC with Subtyping
 - Subtype Relation
 - Variance
 - Algorithmic Typing Rules
- Other Extensions
 - Exceptions
 - Records
 - Conditional Expressions
- Bounded Quantification (Kernel/Full $F_{<:}$)

1. Existential Types

Syntax and Dynamic Semantics

Well-Formedness and Typing Rules

Type Equivalence

Examples

Type Erasure

2. Subtyping for Existential Types

3. Encoding via Universal Types

1. Existential Types

Syntax and Dynamic Semantics

Well-Formedness and Typing Rules

Type Equivalence

Examples

Type Erasure

2. Subtyping for Existential Types

3. Encoding via Universal Types

Recall that **universal types** $(\forall\alpha.\tau)$ describe terms for **all** types α :

$$\text{id} \triangleq \Lambda\alpha. \lambda x : \alpha. x \quad : \quad \forall\alpha. \alpha \rightarrow \alpha$$

In contrast, **existential types** $(\exists\alpha.\tau)$ describe terms whose internal type α is **hidden** from the outside.

For example, the following existential type describes a **record** with two fields x and f , where the type α of x is hidden:

$$\exists\alpha. \{x : \alpha, f : \alpha \rightarrow \text{num}\}$$

This means “*there exists some type α such that the term is a record with a field x of type α and a field f of type $\alpha \rightarrow \text{num}$ ”.*

Existential types are useful for:

- **Information hiding** – the concrete representation of α is hidden.
- **Abstract data types (ADTs)** – only the public interface is exposed.
- **Modules** – a module exposes its interface but hides its implementation.

Let's extend System F with existential types:

Expressions $e ::= \dots \mid \text{pack } \{\tau, e\} \text{ as } \tau \mid \text{let } \{\alpha, x\} = \text{unpack } e \text{ in } e$

Values $v ::= \dots \mid \text{pack } \{\tau, v\} \text{ as } \tau$

Types $\tau ::= \dots \mid \exists \alpha. \tau$

$$\boxed{e \rightarrow e}$$

$$\frac{}{\text{let } \{\alpha, x\} = \text{unpack } (\text{pack } \{\tau, v\} \text{ as } \tau') \text{ in } e \rightarrow e[\alpha \mapsto \tau][x \mapsto v]}$$

$E ::= \dots \mid \text{pack } \{\tau, E\} \text{ as } \tau \mid \text{let } \{\alpha, x\} = \text{unpack } E \text{ in } e$

We extend the well-formedness of types with the existential type:

$$\boxed{\Gamma \vdash \tau} \quad \frac{\Gamma[\alpha] \vdash \tau}{\Gamma \vdash \exists \alpha. \tau}$$

The **introduction rule** (pack) hides the concrete type τ behind α , and the **elimination rule** (unpack) opens the package by introducing a **fresh** type variable α and term variable x :

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash \tau \quad \Gamma \vdash e : \tau'[\alpha \mapsto \tau]}{\Gamma \vdash \text{pack } \{\tau, e\} \text{ as } \exists \alpha. \tau' : \exists \alpha. \tau'}$$

$$\frac{\alpha \notin \text{dom}(\Gamma) \quad \Gamma \vdash e_1 : \exists \alpha. \tau' \quad \Gamma[\alpha][x : \tau'] \vdash e_2 : \tau_2 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \text{let } \{\alpha, x\} = \text{unpack } e_1 \text{ in } e_2 : \tau_2}$$

The side condition $\Gamma \vdash \tau_2$ (with the freshness of α) is essential: the hidden type α **must not escape** the scope of the unpack expression.

Two existential types differ only in the name of the bound variable should be considered equivalent. We extend the type equivalence relation:

$$\boxed{\tau \equiv \tau}$$

$$\frac{\tau \equiv \tau'[\beta \mapsto \alpha] \quad \alpha \notin \text{fvs}(\tau')}{\exists\alpha.\tau \equiv \exists\beta.\tau'}$$

For example,

$$\exists\alpha.\{\mathbf{x} : \alpha, \mathbf{f} : \alpha \rightarrow \mathbf{num}\} \equiv \exists\beta.\{\mathbf{x} : \beta, \mathbf{f} : \beta \rightarrow \mathbf{num}\}$$

by α -conversion of the bound type variable.

Example: Counter ADT

Let's define an abstract counter type:

$$\text{Counter} \triangleq \exists \alpha. \{ \text{new} : \alpha, \text{inc} : \alpha \rightarrow \alpha, \text{get} : \alpha \rightarrow \text{num} \}$$

An implementation using num:

```
counter  $\triangleq$  pack {num, {  
  new=0,  
  inc= $\lambda x : \text{num}. x+1$ ,  
  get= $\lambda x : \text{num}. x$   
} } as Counter
```

A client cannot rely on the concrete representation:

```
let { $\alpha, c$ } = unpack counter in  
c.get (c.inc (c.inc c.new))
```

This evaluates to 2, but the client only knows α abstractly.

We can replace the implementation with a different concrete type $\{\mathbf{n} : \mathbf{num}\}$:

```
counter'  $\triangleq$  pack { $\{\mathbf{n} : \mathbf{num}\}$ , {  
  new= $\{\mathbf{n}=0\}$ ,  
  inc= $\lambda x : \{\mathbf{n} : \mathbf{num}\}. \{\mathbf{n}=x.\mathbf{n}+1\}$ ,  
  get= $\lambda x : \{\mathbf{n} : \mathbf{num}\}. x.\mathbf{n}$   
} } as Counter
```

Both `counter` and `counter'` have the **same type** `Counter`, even though their internal representations differ.

This is the essence of **information hiding**: clients depend only on the interface, not on the implementation.

In Scala, **abstract type members** play the role of existentially quantified types:

```
trait Counter:
  type T                // hidden representation
  val init: T
  def inc(x: T): T
  def get(x: T): Int

val counter: Counter = new Counter:
  type T = Int          // concrete choice, hidden from clients
  val init = 0
  def inc(x: T) = x + 1
  def get(x: T) = x

val n = counter.get(counter.inc(counter.inc(counter.init))) // 2
```

The trait Counter corresponds to

$$\exists \alpha. \{ \text{init} : \alpha, \text{inc} : \alpha \rightarrow \alpha, \text{get} : \alpha \rightarrow \text{num} \}.$$

Alternatively, we can use a **type parameter** and the **wildcard** ?:

```
trait Counter[T]:  
  val init: T  
  def inc(x: T): T  
  def get(x: T): Int  
  
val intCounter: Counter[Int] = new Counter[Int]:  
  val init = 0  
  def inc(x: Int) = x + 1  
  def get(x: Int) = x  
  
val counter: Counter[?] = intCounter // existential: hide T  
val n = counter.get(counter.inc(counter.inc(counter.init)))
```

Here `Counter[?]` corresponds to $\exists \alpha. \text{Counter}[\alpha]$, the same existential type as before, but the abstraction is introduced at the **use site** rather than at the trait definition.

	Abstract type member	Type param + wildcard
Hiding done at	trait definition	use site
Existential is	Counter itself	Counter[?]
Typical style	ML/SML modules	Java generics

Both patterns express the same existential type, but they differ in **where** the abstraction is introduced: the abstract type member hides α inside the trait, while the wildcard ? hides α at the variable's declared type.

We can erase all type information for existential types as well:

$$\text{erase}(\text{pack } \{\tau, e\} \text{ as } \tau') \triangleq \text{erase}(e)$$

$$\text{erase}(\text{let } \{\alpha, x\} = \text{unpack } e_1 \text{ in } e_2) \triangleq \text{let } x = \text{erase}(e_1) \text{ in } \text{erase}(e_2)$$

At runtime, `pack` and `unpack` have **no operational effect**: they only serve as type-checking boundaries.

1. Existential Types

Syntax and Dynamic Semantics

Well-Formedness and Typing Rules

Type Equivalence

Examples

Type Erasure

2. Subtyping for Existential Types

3. Encoding via Universal Types

When the type system has **subtyping**, we can extend the subtype relation to existential types. The body of \exists is **covariant**:

$$\frac{\Gamma[\alpha] \vdash \tau_1 <: \tau_2}{\Gamma \vdash \exists\alpha.\tau_1 <: \exists\alpha.\tau_2}$$

Combined with record subtyping, a richer ADT signature is a subtype of a more permissive one. For example, a counter that exposes `new`, `inc`, and `get` is a subtype of one that only exposes `get`:

$$\exists\alpha.\{\text{new} : \alpha, \text{inc} : \alpha \rightarrow \alpha, \text{get} : \alpha \rightarrow \text{num}\} <: \exists\alpha.\{\text{get} : \alpha \rightarrow \text{num}\}$$

We can support **bounded existential types**, which constrain the hidden type α to be a **subtype** of an upper bound τ :

Expressions $e ::= \dots \mid \text{pack } \{\tau, e\} \text{ as } \exists \alpha <: \tau . \tau$
 $\mid \text{let } \{\alpha <: \tau, x\} = \text{unpack } e \text{ in } e$

Types $\tau ::= \dots \mid \exists \alpha <: \tau . \tau$

The **introduction rule** additionally requires the concrete type to be within the bound, and the **elimination rule** adds the bound to the environment when typing the body:

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash e : \tau''[\alpha \mapsto \tau']}{\Gamma \vdash \text{pack } \{\tau', e\} \text{ as } \exists \alpha <: \tau . \tau'' : \exists \alpha <: \tau . \tau''}$$

$$\frac{\alpha \notin \text{dom}(\Gamma) \quad \Gamma \vdash e_1 : \exists \alpha <: \tau . \tau' \quad \Gamma[\alpha <: \tau][x : \tau'] \vdash e_2 : \tau_2 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \text{let } \{\alpha <: \tau, x\} = \text{unpack } e_1 \text{ in } e_2 : \tau_2}$$

Following the **full** $F_{<}$ approach, we allow **different bounds** on the two sides; the body is checked under the **tighter** bound τ_1 :

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma[\alpha <: \tau_1] \vdash \tau'_1 <: \tau'_2}{\Gamma \vdash (\exists \alpha <: \tau_1.\tau'_1) <: (\exists \alpha <: \tau_2.\tau'_2)}$$

Note the **duality** with \forall : the bound position is **covariant** (opposite of \forall which was contravariant), and the body position is **covariant** (same as \forall). For example, a tighter bound gives a more refined existential:

$$\frac{\vdash \text{num} <: \top \quad [\alpha <: \text{num}] \vdash \alpha <: \alpha}{\vdash (\exists \alpha <: \text{num}.\alpha) <: (\exists \alpha <: \top.\alpha)}$$

This corresponds directly to **wildcards with upper bounds** in Java/Scala:

```
val xs: List[? <: Number] = ... // Scala
List[? extends Number] xs = ...; // Java
```

1. Existential Types

Syntax and Dynamic Semantics

Well-Formedness and Typing Rules

Type Equivalence

Examples

Type Erasure

2. Subtyping for Existential Types

3. Encoding via Universal Types

How can we encode **existential types** using only **universal types**?

Recall that a value of type $\exists\alpha.\tau$ packages a **hidden type** τ' and a **value** of type $\tau[\alpha \mapsto \tau']$, and the only way to use it is via `unpack` – whose body must **not depend on** α .

Idea: instead of carrying the hidden type, become a function that accepts a **callback** ready for **any** α :

$$\exists\alpha.\tau \quad \triangleq \quad \forall\beta. \underbrace{(\forall\alpha.\tau \rightarrow \beta)}_{\text{callback}} \rightarrow \beta$$

Reading the parts:

- β – the result type the consumer wants
- $\forall\alpha.\tau \rightarrow \beta$ – a callback handling **any** α together with a value of type τ , producing β
- the whole – a function taking such a callback, returning β

Pack: hand the hidden type and value to whatever callback is given:

$$\text{pack } \{\tau, e\} \text{ as } \exists\alpha.\tau' \triangleq \Lambda\beta. \lambda f : \forall\alpha.\tau' \rightarrow \beta. f[\tau] e$$

Since f is quantified over $\forall\alpha$, the caller cannot observe which τ we pick.

Unpack: assuming $e_1 : \exists\alpha.\tau'$, pass the body e_2 as a callback:

$$\text{let } \{\alpha, x\} = \text{unpack } e_1 \text{ in } e_2 \triangleq e_1[\tau_2] (\Lambda\alpha. \lambda x : \tau'. e_2)$$

Here τ_2 is the result type of e_2 . The callback $\Lambda\alpha.\lambda x : \tau'. e_2$ is generic over α , matching what `pack` demands.

Let's verify that encoded `unpack(pack ...)` reduces to the expected result:

$$\begin{aligned}
 & (\Lambda\beta. \lambda f. f[\tau] v)[\tau_2] (\Lambda\alpha. \lambda x. e_2) \\
 \rightarrow & (\lambda f. f[\tau] v) (\Lambda\alpha. \lambda x. e_2) \quad (\text{by } \beta \mapsto \tau_2) \\
 \rightarrow & (\Lambda\alpha. \lambda x. e_2)[\tau] v \quad (\text{by } f \mapsto \Lambda\alpha.\lambda x.e_2) \\
 \rightarrow & (\lambda x. e_2[\alpha \mapsto \tau]) v \quad (\text{by } \alpha \mapsto \tau) \\
 \rightarrow & e_2[\alpha \mapsto \tau][x \mapsto v] \quad (\text{by } x \mapsto v)
 \end{aligned}$$

This matches exactly the original `unpack-of-pack` rule:

$$\text{let } \{\alpha, x\} = \text{unpack } (\text{pack } \{\tau, v\} \text{ as } \tau') \text{ in } e_2 \rightarrow e_2[\alpha \mapsto \tau][x \mapsto v]$$

Therefore, System F with universal types **alone** is sufficient to encode existential types – they add no expressiveness at the type level.

- Type Operators

Jihyeok Park
jihyeok_park@korea.ac.kr
<https://plrg.korea.ac.kr>