Lecture 19 – Existential Types

AAA551: Programming Language Theory

Jihyeok Park

2026 Spring

Recall

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

Contents

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

Contents

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

Motivation

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

$$\texttt{id} \triangleq \Lambda\alpha.\ \lambda x : \alpha.\ x \qquad : \qquad \forall\alpha.\alpha \to \alpha$$

Motivation

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

$$\texttt{id} \triangleq \Lambda\alpha.\ \lambda x : \alpha.\ x \qquad : \qquad \forall\alpha.\alpha \to \alpha$$

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

Motivation

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

$$\texttt{id} \triangleq \Lambda\alpha.\ \lambda x : \alpha.\ x \qquad : \qquad \forall\alpha.\alpha \to \alpha$$

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:

$$\exists\alpha.\{\texttt{x} : \alpha,\ \texttt{f} : \alpha \to \texttt{num}\}$$

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 $\alpha \to num$"*.

Motivation

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

$$\texttt{id} \triangleq \Lambda\alpha.\ \lambda x : \alpha.\ x \qquad : \qquad \forall\alpha.\alpha \to \alpha$$

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:

$$\exists\alpha.\{\texttt{x} : \alpha,\ \texttt{f} : \alpha \to \texttt{num}\}$$

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 $\alpha \to num$"*.

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.

Syntax and Dynamic Semantics

Let's extend System F with existential types:

Expressions	$e ::= \ldots \mid \texttt{pack}\ \{\tau, e\}\ \texttt{as}\ \tau \mid \texttt{let}\ \{\alpha, x\} = \texttt{unpack}\ e\ \texttt{in}\ e$
Values	$v ::= \ldots \mid \texttt{pack}\ \{\tau, v\}\ \texttt{as}\ \tau$
Types	$\tau ::= \ldots \mid \exists \alpha.\tau$

Syntax and Dynamic Semantics

Let's extend System F with existential types:

Expressions	$e ::= \ldots \mid \texttt{pack}\ \{\tau, e\}\ \texttt{as}\ \tau \mid \texttt{let}\ \{\alpha, x\} = \texttt{unpack}\ e\ \texttt{in}\ e$
Values	$v ::= \ldots \mid \texttt{pack}\ \{\tau, v\}\ \texttt{as}\ \tau$
Types	$\tau ::= \ldots \mid \exists \alpha.\tau$

$$\boxed{e \to e}$$

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

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

Well-Formedness and Typing Rules

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

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

Well-Formedness and Typing Rules

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

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

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:

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

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

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

Well-Formedness and Typing Rules

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

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

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:

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

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

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

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.

Type Equivalence

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

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

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

Type Equivalence

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

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

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

For example,

$$\exists \alpha.\{\mathtt{x} : \alpha,\ \mathtt{f} : \alpha \to \mathtt{num}\} \ \equiv \ \exists \beta.\{\mathtt{x} : \beta,\ \mathtt{f} : \beta \to \mathtt{num}\}$$

by α-conversion of the bound type variable.

Example: Counter ADT

Let's define an abstract counter type:

$$\texttt{Counter} \triangleq \exists\alpha.\{\texttt{new} : \alpha,\ \texttt{inc} : \alpha \to \alpha,\ \texttt{get} : \alpha \to \texttt{num}\}$$

Example: Counter ADT

Let's define an abstract counter type:

$$\texttt{Counter} \triangleq \exists\alpha.\{\texttt{new} : \alpha,\ \texttt{inc} : \alpha \to \alpha,\ \texttt{get} : \alpha \to \texttt{num}\}$$

An implementation using `num`:

$$\begin{array}{l}\texttt{counter} \triangleq \texttt{pack}\ \{\texttt{num},\ \{\\ \quad \texttt{new=}0,\\ \quad \texttt{inc=}\lambda x : \texttt{num}.\ x\texttt{+}1,\\ \quad \texttt{get=}\lambda x : \texttt{num}.\ x\\ \}\ \}\ \texttt{as Counter}\end{array}$$

Example: Counter ADT

PLRG

Let's define an abstract counter type:

$$\texttt{Counter} \triangleq \exists\alpha.\{\texttt{new} : \alpha,\ \texttt{inc} : \alpha \to \alpha,\ \texttt{get} : \alpha \to \texttt{num}\}$$

An implementation using `num`:

$$\begin{array}{l}\texttt{counter} \triangleq \texttt{pack}\ \{\texttt{num},\ \{\\ \quad \texttt{new=}0,\\ \quad \texttt{inc=}\lambda x : \texttt{num}.\ x\texttt{+}1,\\ \quad \texttt{get=}\lambda x : \texttt{num}.\ x\\ \}\ \}\ \texttt{as Counter}\end{array}$$

A client cannot rely on the concrete representation:

$$\begin{array}{l}\texttt{let}\ \{\alpha, c\} = \texttt{unpack counter in}\\ c.\texttt{get}\ (c.\texttt{inc}\ (c.\texttt{inc}\ c.\texttt{new}))\end{array}$$

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

Example: Alternative Implementation

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

$$
\begin{array}{l}
\texttt{counter}' \triangleq \texttt{pack}\ \{\{\texttt{n} : \texttt{num}\},\ \{ \\
\quad \texttt{new}=\{\texttt{n}=0\}, \\
\quad \texttt{inc}=\lambda x : \{\texttt{n} : \texttt{num}\}.\ \{\texttt{n}=x.\texttt{n}+1\}, \\
\quad \texttt{get}=\lambda x : \{\texttt{n} : \texttt{num}\}.\ x.\texttt{n} \\
\}\ \}\ \texttt{as Counter}
\end{array}
$$

Example: Alternative Implementation

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

$$\begin{array}{l}\texttt{counter}' \triangleq \texttt{pack}\ \{\{\texttt{n} : \texttt{num}\},\ \{ \\ \quad \texttt{new}=\{\texttt{n}=0\}, \\ \quad \texttt{inc}=\lambda x : \{\texttt{n} : \texttt{num}\}.\ \{\texttt{n}=x.\texttt{n}+1\}, \\ \quad \texttt{get}=\lambda x : \{\texttt{n} : \texttt{num}\}.\ x.\texttt{n} \\ \}\ \}\ \texttt{as}\ \texttt{Counter}\end{array}$$

Both `counter` and $\texttt{counter}'$ have the **same type** `Counter`, even though their internal representations differ.

Example: Alternative Implementation

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

$$
\begin{aligned}
&\texttt{counter}' \triangleq \texttt{pack}\ \{\{\texttt{n} : \texttt{num}\},\ \{ \\
&\quad \texttt{new=}\{\texttt{n=}0\}, \\
&\quad \texttt{inc=}\lambda x : \{\texttt{n} : \texttt{num}\}.\ \{\texttt{n=}x.\texttt{n+}1\}, \\
&\quad \texttt{get=}\lambda x : \{\texttt{n} : \texttt{num}\}.\ x.\texttt{n} \\
&\}\ \}\ \texttt{as Counter}
\end{aligned}
$$

Both `counter` and $\texttt{counter}'$ 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.

Counter in Scala – Abstract Type Member

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
```

Counter in Scala – Abstract Type Member

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
$\exists\alpha.\{\texttt{init} : \alpha,\ \texttt{inc} : \alpha \to \alpha,\ \texttt{get} : \alpha \to \texttt{num}\}$.

Counter in Scala – Type Parameter + Wildcard

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)))
```

Counter in Scala – Type Parameter + Wildcard

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

Two Patterns for Existential Types in Scala

	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.

Type Erasure

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

$$\begin{aligned} \texttt{erase}(\texttt{pack}\ \{\tau, e\}\ \texttt{as}\ \tau') &\triangleq \texttt{erase}(e) \\ \texttt{erase}(\texttt{let}\ \{\alpha, x\} = \texttt{unpack}\ e_1\ \texttt{in}\ e_2) &\triangleq \texttt{let}\ x = \texttt{erase}(e_1)\ \texttt{in}\ \texttt{erase}(e_2) \end{aligned}$$

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

Contents

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

Subtyping for Existential Types

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

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

Subtyping for Existential 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`:

$$\exists\alpha.\{\texttt{new} : \alpha,\ \texttt{inc} : \alpha \to \alpha,\ \texttt{get} : \alpha \to \texttt{num}\} <: \exists\alpha.\{\texttt{get} : \alpha \to \texttt{num}\}$$

Bounded Existential Types

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

$$\begin{array}{lll} \text{Expressions} & e ::= \ldots \mid \texttt{pack}\ \{\tau, e\}\ \texttt{as}\ \exists \alpha <: \tau\ .\tau \\ & \quad \mid \texttt{let}\ \{\alpha <: \tau\ , x\} = \texttt{unpack}\ e\ \texttt{in}\ e \\ \text{Types} & \tau ::= \ldots \mid \exists \alpha <: \tau\ .\tau \end{array}$$

Bounded Existential Types

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

$$\begin{array}{lll} \text{Expressions} & e ::= \ldots \mid \texttt{pack}\,\{\tau, e\}\,\texttt{as}\,\exists\alpha <: \tau\,.\tau \\ & \quad \mid \texttt{let}\,\{\alpha <: \tau\,, x\} = \texttt{unpack}\,e\,\texttt{in}\,e \\ \text{Types} & \tau ::= \ldots \mid \exists\alpha <: \tau\,.\tau \end{array}$$

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:

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

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

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

Bounded Existential Types – Subtype Relation

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

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

Note the **duality** with $\forall$: the bound position is **covariant** (opposite of $\forall$ which was contravariant), and the body position is **covariant** (same as $\forall$).

Bounded Existential Types – Subtype Relation

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

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

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:

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

Bounded Existential Types – Subtype Relation

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

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

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:

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

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

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

Contents

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

Encoding Existential Types – Idea

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

Encoding Existential Types – Idea

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.

Encoding Existential Types – Idea

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:

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

Encoding Existential Types – Idea

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:

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

Reading the parts:

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

Encoding Existential Types – Pack and Unpack

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

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

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

Encoding Existential Types – Pack and Unpack

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

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

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:

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

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.

Encoding Existential Types – Reduction Check

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

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

Encoding Existential Types – Reduction Check

Let's verify that encoded $\texttt{unpack}(\texttt{pack}\ \ldots)$ reduces to the expected result:

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

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

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

Encoding Existential Types – Reduction Check

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

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

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

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

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

Next Lecture

- Type Operators

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