

Today, we already looked at encoding encoding functions using Church notation (regular class time) and at the relationship between universal and existential types (Make-Up Lecture 1). Now we will discuss the use of existential types and specify introduction and elimination rules for them.

1 Why do we need existential types?

In the past, we have specified Church numerals using the type

$$CNat = \forall X.(X \longrightarrow X) \longrightarrow X \longrightarrow X$$

Whenever a term matches this type, i.e. it is, for some type X , a function whose argument is a function from X to X , and whose return type is a function with argument and return types X , then it could be considered a $CNat$.

The $CNat$ was in lazy in a way: The formulation said “If you give me a successor function and a zero value, I’ll give you the value.” The problem is that the user of a $CNat$ specifies the successor function and the zero value, and the creator of the $CNat$ framework has no control over what these are. The successor function could, for example, diverge or have side effects.

We could provide functions, one might call them “factories”, for the successor function and the zero value:

$$\begin{array}{ll} NatCNat = CNat[Nat] & NatCNat = (Nat \longrightarrow Nat) \longrightarrow Nat \longrightarrow Nat \\ makeSucc = \lambda x : Nat. succ(x) & makeSucc : Nat \longrightarrow Nat \\ makeZero = 0 & makeZero : Nat \end{array}$$

Then users of our framework could build Church numerals using these factories, but it still does not force them into using them. As long as they provide a successor function or a zero value with the same types, they can be used.

If we could hide away the actual type of the X used, then users would not know what types their own successor function or zero value should have, and that way, they would have to use the provided ones, because only they can create terms of the proper type. Existential types provide this facility.

The type of a Church numeral using existential types can be written the following way:

$$\exists X.(X \longrightarrow X, X)$$

2 Adding Existential Types

To support existential types, we make changes to our syntactic forms for expressions, types, and values¹:

$$\begin{aligned} e &::= 0 \mid \text{succ } e \mid x \mid \lambda x : t. e \mid ee \mid \lambda X. e \mid e[t] \mid \text{pack } t \text{ as } X \text{ in } e \mid \text{let}_{\exists} x = e_1 \text{ in } e_2 \\ t &::= X \mid t \longrightarrow t \mid \forall X. t \mid \exists X. t \\ v &::= 0 \mid \text{succ } v \mid \lambda x. t \mid \lambda X. t \mid \text{pack } t \text{ as } X \text{ in } e \end{aligned}$$

`pack t_1 as X in e` means that we take an expression e with type t_2 , which may use the type t_1 , and package it up as an existential type $\exists X. (t_2[t_1 := X])$ where all occurrences of t_1 have been replaced with X . We could, for example, do the following²:

$$\text{pack } Nat \text{ as } Y \text{ in } \lambda x : Nat. \text{succ } x$$

¹The textbook “Types and Programming Languages” (TAPL) by Benjamin Pierce uses a different syntax: `{* t, e } as t` corresponds to our `pack t as X in e` construct.

²In TAPL, this would be `{* $Nat, \lambda x : Nat. \text{succ } x$ } as $(\exists Y. Y \longrightarrow Y)$` .

This would have type $\exists Y. Y \longrightarrow Y$. The fact that we have used type *Nat* has been hidden. The knowledge that it was *Nat* only exists inside the scope introduced by `pack`. Anything that requires this knowledge, e.g. the factories, has to be written inside this scope as well. For convenience, the factories can be provided as members of a record:

$$\begin{aligned} & \text{myNat} = \\ \text{pack Nat as } X \text{ in } & \{ \text{makeSucc} = (\lambda x : \text{Nat}. \text{succ}(x)), \text{makeZero} = 0 \} : \\ & \exists X. \{ \text{makeSucc} : X \longrightarrow X, \text{makeZero} : X \} \end{aligned}$$

To make use of *makeSucc* and *makeZero* now, we can use the existential `let∃ x = e1 in e2`. Here, *x* is a variable name, *e₁* is an expression of existential type, and *e₂* is an expression that can use what is inside *e₁*. We can, for example, say

$$\text{let}_{\exists} v = \text{myNat} \text{ in } v.\text{makeZero}$$

3 Typing Rules

To type-check terms with existential types, we need to add introduction and elimination rules.

The introduction rule is

$$\frac{\Gamma \vdash e : t_2[X := t_1]}{\Gamma \vdash \text{pack } t_1 \text{ as } X \text{ in } e : (\exists X. t_2)}$$

It hides the actual type *t₁* that was used and replaces it with the *X* of the existential type.

Our first attempt at writing an elimination rule is

$$\frac{\Gamma \vdash e_1 : (\exists X. t_1) \quad \Gamma, X, (x : t_1) \vdash e_2 : t_2}{\Gamma \vdash \text{let}_{\exists} x = e_1 \text{ in } e_2 : t_2}$$

This rule ensures that e_1 has existential type $\exists X. t_1$ and then type-checks e_2 using an extended typing context $\Gamma, X, (x : t_1)$. The type variable X is added because it may be used inside t_1 . The type of the entire expression is the type of e_2 .

There is one problem with this rule. We can have a term of type $\exists X. X$:

$$problematic = \text{pack } Nat \text{ as } X \text{ in } 0 : \exists X. X$$

When we use such an expression in $\text{let}_{\exists} x = problematic \text{ in } x$, we get the typing derivation

$$\frac{\frac{(problematic : \exists X. X) \in \Gamma}{\Gamma \vdash problematic : (\exists X. X)} \quad \frac{(x : X) \in \Gamma, X, (x : X)}{\Gamma, X, (x : X) \vdash x : X}}{\Gamma \vdash \text{let}_{\exists} x = problematic \text{ in } x : X}$$

The type of the expression is X , yet X does not appear in our typing context Γ . We have leaked the variable of the existential type outside of its scope.

To make sure this cannot happen, we make the restriction that the variable of the existential type X cannot be a free variable of the type of e_2 :

$$\frac{\Gamma \vdash e_1 : (\exists X. t_1) \quad \Gamma, X, (x : t_1) \vdash e_2 : t_2 \quad X \notin FV(t_2)}{\Gamma \vdash \text{let}_{\exists} x = e_1 \text{ in } e_2 : t_2}$$

4 Homework

The following problems on existential types are due on Wednesday, April 27, 2005:

- 24.1.1
- 24.2.1
- 24.2.2
- 24.2.3