

#### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

#### Existentials

Examples  
Metatheory  
Church

## MPRI 2.4

# Algebraic data types and existential types

François Pottier



2024

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

# 1 Data types

Primitive sums, products, and recursive types

Algebraic data types

Scott and Church encodings

# 2 Existential types

Examples

Metatheory

Church encoding

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

# Towards data types

Many data types can be built out of **sums** and **products** and a form of **recursion** at the level of types.

**Binary** sum  $+$  and product  $\times$ , and their **neutral elements** 0 and 1, suffice.

- The **unit** type is 1.
- The **empty** type is 0.
- The **Boolean** type is  $1 + 1$ .
- The type  $\mathbb{N}$  of the natural numbers must satisfy  $\mathbb{N} \simeq 1 + \mathbb{N}$ .
- The type  $\mathbb{L}(X)$  of lists of elements of type  $X$  must satisfy

$$\mathbb{L}(X) \simeq 1 + X \times \mathbb{L}(X)$$

## Three technical approaches to data types

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

There are three main approaches to extending System  $F$  with data types:

- consider  $0, 1, +, \times$ , and recursive types  $\mu X.T$  as **primitive concepts** and encode all data types in terms of these concepts;
- consider **algebraic data types** as primitive and view sums, products, naturals, lists, etc., as instances of this general concept;
- introduce **no new primitive concept** and remark that **inductive types** can be encoded in System  $F$ .

In practice, the second approach is the most natural and user-friendly.

All three approaches, and their connections, are worth understanding.

## Data types

Primitive sums,  
products, and  
recursive types

Algebraic data  
Scott & Church

## Existentials

Examples

Metatheory

Church

# 1 Data types

Primitive sums, products, and recursive types

Algebraic data types

Scott and Church encodings

# 2 Existential types

Examples

Metatheory

Church encoding

## Binary products

It is easy to add **pairs** and **projections** to the (call-by-value)  $\lambda$ -calculus.

$$\begin{aligned} t &::= \dots \mid (t, t) \mid \pi_i t && \text{where } i \in \{1, 2\} \\ v &::= \dots \mid (v, v) \\ E &::= \dots \mid (E, t) \mid (v, E) \mid \pi_i E \end{aligned}$$

One new reduction rule is needed:  $\pi_i (v_1, v_2) \longrightarrow v_i$ .

A new type constructor is needed:

## Binary products

It is easy to add **pairs** and **projections** to the (call-by-value)  $\lambda$ -calculus.

$$\begin{aligned} t &::= \dots \mid (t, t) \mid \pi_i t && \text{where } i \in \{1, 2\} \\ v &::= \dots \mid (v, v) \\ E &::= \dots \mid (E, t) \mid (v, E) \mid \pi_i E \end{aligned}$$

One new reduction rule is needed:  $\pi_i (v_1, v_2) \longrightarrow v_i$ .

A new type constructor is needed:  $T ::= \dots \mid T \times T$ .

Two new typing rules are needed:

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

## Binary products

It is easy to add **pairs** and **projections** to the (call-by-value)  $\lambda$ -calculus.

$$\begin{aligned}t &::= \dots \mid (t, t) \mid \pi_i t && \text{where } i \in \{1, 2\} \\v &::= \dots \mid (v, v) \\E &::= \dots \mid (E, t) \mid (v, E) \mid \pi_i E\end{aligned}$$

One new reduction rule is needed:  $\pi_i (v_1, v_2) \longrightarrow v_i$ .

A new type constructor is needed:  $T ::= \dots \mid T \times T$ .

Two new typing rules are needed:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash \pi_i t : T_i}$$

**Exercise:** extend the proofs of Subject Reduction and Progress.

**Variation:** introduce the elimination form  $\text{let } (x_1, x_2) = t \text{ in } t$ .

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church



## Data types

Primitive sums,  
products, and  
recursive types

Algebraic data

Scott &amp; Church

## Existentials

Examples

Metatheory

Church

The **unit** type 1 can be viewed as a product type of arity 0.

It has an **introduction** form but no **elimination** form.

$$t ::= \dots \mid ()$$
$$v ::= \dots \mid ()$$

– no new evaluation context

No new reduction rule is needed.

A new type constructor is needed:  $T ::= \dots \mid 1$ .

One new typing rule is needed:

$$\Gamma \vdash () : 1$$

**Variation:** introduce the elimination form  $\text{let } () = t \text{ in } t$ .

## Binary sums

Let us add **injections** and a **case analysis** to (call-by-value)  $\lambda$ -calculus.

$$\begin{aligned} t &::= \dots \mid \text{inj}_i t \mid \text{case } t \text{ of } t_1 \parallel t_2 && \text{where } i \in \{1, 2\} \\ v &::= \dots \mid \text{inj}_i v \\ E &::= \dots \mid \text{inj}_i E \mid \text{case } E \text{ of } t_1 \parallel t_2 \end{aligned}$$

One new reduction rule is needed:  $\text{case } \text{inj}_i v \text{ of } t_1 \parallel t_2 \longrightarrow t_i v$ .

In a case construct, the branches  $t_1$  and  $t_2$  should be functions.

A new type constructor is needed:

## Binary sums

Let us add **injections** and a **case analysis** to (call-by-value)  $\lambda$ -calculus.

$$\begin{aligned} t &::= \dots \mid \text{inj}_i t \mid \text{case } t \text{ of } t_1 \parallel t_2 && \text{where } i \in \{1, 2\} \\ v &::= \dots \mid \text{inj}_i v \\ E &::= \dots \mid \text{inj}_i E \mid \text{case } E \text{ of } t_1 \parallel t_2 \end{aligned}$$

One new reduction rule is needed:  $\text{case } \text{inj}_i v \text{ of } t_1 \parallel t_2 \longrightarrow t_i v$ .

In a **case** construct, the branches  $t_1$  and  $t_2$  should be functions.

A new type constructor is needed:  $T ::= \dots \mid T + T$ .

Two new typing rules are needed:

### Data types

Primitive sums,  
products, and  
recursive types

Algebraic data  
Scott & Church

### Existentials

Examples

Metatheory  
Church

## Binary sums

Let us add **injections** and a **case analysis** to (call-by-value)  $\lambda$ -calculus.

$$\begin{aligned} t &::= \dots \mid \text{inj}_i t \mid \text{case } t \text{ of } t_1 \parallel t_2 && \text{where } i \in \{1, 2\} \\ v &::= \dots \mid \text{inj}_i v \\ E &::= \dots \mid \text{inj}_i E \mid \text{case } E \text{ of } t_1 \parallel t_2 \end{aligned}$$

One new reduction rule is needed:  $\text{case } \text{inj}_i v \text{ of } t_1 \parallel t_2 \longrightarrow t_i v$ .

In a **case** construct, the branches  $t_1$  and  $t_2$  should be functions.

A new type constructor is needed:  $T ::= \dots \mid T + T$ .

Two new typing rules are needed:

$$\frac{\Gamma \vdash t : T_i}{\Gamma \vdash \text{inj}_i t : T_1 + T_2} \qquad \frac{\Gamma \vdash t : T_1 + T_2 \quad \Gamma \vdash t_1 : T_1 \rightarrow T' \quad \Gamma \vdash t_2 : T_2 \rightarrow T'}{\Gamma \vdash \text{case } t \text{ of } t_1 \parallel t_2 : T'}$$

**Exercise:** extend the proofs of Subject Reduction and Progress.

## Data types

Primitive sums,  
products, and  
recursive types

Algebraic data  
Scott & Church

## Existentials

Examples

Metatheory  
Church

The **empty** type can be viewed as a sum type of arity 0.

It has an **elimination** form but no **introduction** form.

$$t ::= \dots \mid \textit{absurd } t$$

– no new value

$$E ::= \dots \mid \textit{absurd } E$$

No new reduction rule is needed. *absurd* *v* is stuck.

A new type constructor is needed:  $T ::= \dots \mid 0$ .

One new typing rule is needed:

$$\frac{\Gamma \vdash t : 0}{\Gamma \vdash \textit{absurd } t : T'}$$

**Exercise:** extend the proof of Progress.

## Approaches to recursive types

### Data types

Primitive sums,  
products, and  
recursive types

Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

Recall what was said earlier about **recursive types**:

- Natural numbers must satisfy  $\mathbb{N} \simeq 1 + \mathbb{N}$ .

*A natural number is **either zero**  
or **the successor** of a natural number.*

- Lists must satisfy  $\mathbb{L}(X) \simeq 1 + X \times \mathbb{L}(X)$ .

*A list is **either the empty list**  
or **a pair** of an element and a list.*

The types  $\mathbb{N}$  and  $\mathbb{L}(X)$  appear to satisfy **recursive equations**.

What is  $\simeq$ ? How can the types  $\mathbb{N}$  and  $\mathbb{L}(X)$  be defined?

## Approaches to recursive types

Several answers are possible.

- 1 **Equi-recursive types.** Interpret  $\simeq$  as **equality**. A type is a possibly **infinite tree**. The notation  $\mu X.T$  describes such a tree.
- 2 **Structural iso-recursive types.** Interpret  $\simeq$  as **isomorphism**. A type is a **finite tree**. The syntax of types is extended with a general form of recursive type,  $\mu X.T$ .
- 3 **Nominal iso-recursive types.** Interpret  $\simeq$  as **isomorphism**. A type is a **finite tree**. The syntax of types is extended with user-defined types such as  $\mathbb{N}$ ,  $\mathbb{L}(X)$ , or (more generally) **algebraic data types**.

## Approach 1: equi-recursive types

Suppose we want  $\mathbb{N} = 1 + \mathbb{N}$  and  $\mathbb{L}(X) = 1 + X \times \mathbb{L}(X)$ .

Then, a type must be a **possibly infinite tree**.

```
CoInductive ty :=  
  | TyVar (x : var)  
  | TyFun (A B : ty).
```

Here is an example of an infinite tree:

```
CoFixpoint arrows :=  
  TyFun arrows arrows.
```

On paper, this type is usually written  $\mu X. X \rightarrow X$ .

$\mu$  is **not a constructor** in the syntax of types.

The equality  $arrows = arrows \rightarrow arrows$  is true.

In Coq, a suitable notion of extensional equality of types  
must be co-inductively defined.



## Approach 1: equi-recursive types

### Data types

Primitive sums,  
products, and  
recursive types

Algebraic data

Scott & Church

### Existentials

Examples

Metatheory

Church

In this approach, assuming we have sum and product types,

- $\mathbb{N}$  can be defined as a notation for  $\mu X. 1 + X$ ,
- $\mathbb{L}(X)$  can be defined as a notation for  $\mu Y. 1 + X \times Y$ .

In this approach,

- $\text{inj}_1 ()$  has type  $\mathbb{N}$ , and also has type  $\mathbb{L}(\mathbb{N})$ .

This works in theory, but is not very pleasant in practice.

## Approach 1: equi-recursive types

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

In this approach, **only the nature of types changes**,  
from finite trees to possibly infinite trees.

The typing rules of the simply-typed  $\lambda$ -calculus,  
or of System  $F$ , are unchanged.

The proof of type soundness is unchanged.

**Exercise:** on paper or in Coq, extend the simply-typed  $\lambda$ -calculus with equi-recursive types, and update the proof of type soundness, where needed. Prove that every (pure, closed)  $\lambda$ -term has type  $\mu X. X \rightarrow X$ .

## Approach 1: equi-recursive types

In this approach, many nonsensical terms become well-typed.

```
ocaml -rectypes
# let f x = [x] :: x;;
val f : (('a list as 'b) list as 'a) -> 'b list = <fun>
```

OCaml infers that  $f$  has type  $A \rightarrow \mathbb{L}(B)$   
where  $\mathbb{L}(B) = A$  and  $\mathbb{L}(A) = B$ .

This type is in fact equal to  $lists \rightarrow lists$ ,  
where  $lists = \mu X. \mathbb{L}(X) = \mathbb{L}(lists) = \mathbb{L}(\mathbb{L}(\dots))$ .

```
# type lists = ('a list as 'a);;
type lists = 'a list as 'a
# let f (x : lists) : lists = [x] :: x;;
val f : lists -> lists = <fun>
```

This downside explains why this approach is not used in practice.

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

## Approach 2: structural iso-recursive types

Suppose we want types to remain **finite** trees.

We **extend the syntax of types**:  $T ::= \dots \mid \mu X. T$ .

We **extend the syntax of terms** with introduction and elimination forms:

$$\begin{aligned} t &::= \dots \mid \text{fold}_{\mu X. T} \, t \mid \text{unfold}_{\mu X. T} \, t \\ v &::= \dots \mid \text{fold}_{\mu X. T} \, v \\ E &::= \dots \mid \text{fold}_{\mu X. T} \, E \mid \text{unfold}_{\mu X. T} \, E \end{aligned}$$

Their operational semantics is simple:

$$\text{unfold}_{\mu X. T} (\text{fold}_{\mu X. T} \, v) \longrightarrow v$$

Two new typing rules are introduced:

$$\frac{\Gamma \vdash t : T[\mu X. T / X]}{\Gamma \vdash \text{fold}_{\mu X. T} \, t : \mu X. T} \qquad \frac{\Gamma \vdash t : \mu X. T}{\Gamma \vdash \text{unfold}_{\mu X. T} \, t : T[\mu X. T / X]}$$

## Approach 2: structural iso-recursive types

### Data types

Primitive sums,  
products, and  
recursive types

Algebraic data

Scott & Church

### Existentials

Examples

Metatheory

Church

$fold_{\mu X.T}$  and  $unfold_{\mu X.T}$  are **coercions**  
between the types  $\mu X.T$  and  $T[\mu X.T/X]$ .  
They are mutual inverses.

These types are said to be **isomorphic**:

$$\mu X.T \simeq T[\mu X.T/X]$$

**Exercise:** on paper or in Coq, extend the simply-typed  $\lambda$ -calculus with iso-recursive types. Update the proof of type soundness where needed.

## Approach 2: structural iso-recursive types

### Data types

Primitive sums,  
products, and  
recursive types

Algebraic data  
Scott & Church

### Existentials

Examples

Metatheory

Church

In this approach, as in the previous approach,

- $\mathbb{N}$  can be defined as a notation for  $\mu X. 1 + X$ ,
- $\mathbb{L}(X)$  can be defined as a notation for  $\mu Y. 1 + X \times Y$ .

In this approach,

- $\text{inj}_1 ()$  has type  $1 + \mathbb{N}$ , and also has type  $1 + \mathbb{N} \times \mathbb{L}(\mathbb{N})$ ,
- $\text{fold}_{\mathbb{N}} (\text{inj}_1 ())$  has type  $\mathbb{N}$ .
- $\text{fold}_{\mathbb{L}(\mathbb{N})} (\text{inj}_1 ())$  has type  $\mathbb{L}(\mathbb{N})$ .

This works in theory, but is not very pleasant in practice.

## Approach 3: nominal iso-recursive types

Let us view  $\mathbb{N}$  as a **primitive type**:  $T ::= \dots \mid \mathbb{N}$ .

Give new typing rules—two introduction rules and an elimination rule:

$$\begin{array}{c}
 \frac{\Gamma \vdash t : 1}{\Gamma \vdash \text{inj}_1 t : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \text{inj}_2 t : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N} \quad \Gamma \vdash t_1 : 1 \rightarrow T' \quad \Gamma \vdash t_2 : \mathbb{N} \rightarrow T'}{\Gamma \vdash \text{case } t \text{ of } t_1 \parallel t_2 : T'}
 \end{array}$$

These are **exactly the typing rules proposed earlier for binary sums** where we have replaced  $T_1 + T_2$  with  $\mathbb{N}$ ,  $T_1$  with  $1$ , and  $T_2$  with  $\mathbb{N}$ .

We have  $\mathbb{N} \simeq 1 + \mathbb{N}$ : one can write  $\text{in} : 1 + \mathbb{N} \rightarrow \mathbb{N}$  and  $\text{out} : \mathbb{N} \rightarrow 1 + \mathbb{N}$  such that  $\text{in} \cdot \text{out} \equiv_{\beta\eta} \text{out} \cdot \text{in} \equiv_{\beta\eta} \text{id}$ . This is an **iso-recursive** approach.

In this approach, there is no  $\mu$  syntax or  $\mu$  notation.

$\mathbb{N}$  is viewed as the **name** of a basic type.

$\mathbb{N}$  is an **abstract** type with construction and deconstruction operations.

## Approach 3: nominal iso-recursive types

Let us view  $\mathbb{L}(X)$  as a **primitive type constructor**:  $T ::= \dots \mid \mathbb{L}(T)$ .

Give new typing rules—two introduction rules and an elimination rule:

$$\frac{\Gamma \vdash t : 1}{\Gamma \vdash \text{inj}_1 t : \mathbb{L}(T)} \qquad \frac{\Gamma \vdash t : T \times \mathbb{L}(T)}{\Gamma \vdash \text{inj}_2 t : \mathbb{L}(T)}$$

$$\frac{\Gamma \vdash t : \mathbb{L}(T) \quad \Gamma \vdash t_1 : 1 \rightarrow T' \quad \Gamma \vdash t_2 : T \times \mathbb{L}(T) \rightarrow T'}{\Gamma \vdash \text{case } t \text{ of } t_1 \parallel t_2 : T'}$$

These are again **exactly the typing rules of binary sums** where we have replaced  $T_1 + T_2$  with  $\mathbb{L}(X)$ ,  $T_1$  with  $1$ , and  $T_2$  with  $X \times \mathbb{L}(X)$ .

We have  $\mathbb{L}(X) \simeq 1 + X \times \mathbb{L}(X)$ .

$\mathbb{L}$  is viewed as the **name** of a basic type constructor.

$\mathbb{L}(X)$  is an **abstract** type with construction and deconstruction operations.



## Data types

Primitive sums,  
products, and  
recursive types

### Algebraic data

Scott & Church

## Existentials

Examples

Metatheory

Church

# 1 Data types

Primitive sums, products, and recursive types

**Algebraic data types**

Scott and Church encodings

# 2 Existential types

Examples

Metatheory

Church encoding

## Data types

Primitive sums,  
products, and  
recursive types

Algebraic data  
Scott & Church

## Existentials

Examples

Metatheory  
Church

# Algebraic data types

Instead of offering a fixed set of primitive types such as  $\mathbb{N}$  and  $\mathbb{L}(X)$ ,  
let users define whatever custom types they need  
using sums and products (of arbitrary arity) and recursion.

This idea gives rise to algebraic data types.

```
type      nat = Zero | Succ of nat
type 'a list = Nil   | Cons of 'a * 'a list
type 'a tree = Leaf  | Node of 'a tree * 'a * 'a tree
```

# Algebraic data types

## Data types

Primitive sums,  
products, and  
recursive types

Algebraic data

Scott & Church

## Existentials

Examples

Metatheory

Church

It is now easy to **construct** data:

```
let one : nat = Succ Zero
```

and to **deconstruct** data:

```
let predecessor (n : nat) : nat =  
  match n with  
  | Zero -> Zero  
  | Succ n -> n
```

OCaml also offers a more concise function definition form:

```
let predecessor : nat -> nat =  
  function Zero -> Zero | Succ n -> n
```

## Algebraic data types

Pattern matching allows deconstructing data in depth.

This is an implementation of rotations of binary trees in Standard ML:

```
fun n (v, l, r) =
  T(v, 1 + size l + size r, l, r)
fun single_L (a, x, T(b, _, y, z)) =
  n(b, n(a, x, y), z)
fun double_L (a, x, T(c, _, T(b, _, y1, y2), z)) =
  n(b, n(a, x, y1), n(c, y2, z))
```

It is concise!

That said, it is not perfect. Adopting the convention  $(l, v, r)$  would make it much easier to read and debug.

Adams,  
Efficient sets—a balancing act, 1993.

## Algebraic data types

Named types, named data constructors, and pattern matching make algebraic data types extremely pleasant and safe to use.

be instantiated to any type. We suspect that a great many errors are caused by the complications introduced when encoding data in terms of the commonly-supplied low-level types; the provision of a simple and powerful facility for defining types should greatly simplify the programmer's task.

Burstall, MacQueen, Sannella,  
HOPE: An experimental applicative language, 1980.

# Products and sums as algebraic data types

## Data types

Primitive sums,  
products, and  
recursive types

### Algebraic data

Scott & Church

## Existentials

Examples

Metatheory

Church

Sums and products can be viewed as algebraic data types.

```
type ('a, 'b) sum = Left of 'a | Right of 'b
type void = | (* zero constructors *)
type ('a, 'b) pair = Pair of 'a * 'b
type unit = ()
```

Deconstructing the type void works as expected:

```
let absurd (type a) (x : void) : a =
  match x with _ -> . (* zero branches *)
```

# An isomorphism

## Data types

Primitive sums,  
products, and  
recursive types

Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

The types  $\mathbb{N}$  and  $1 + \mathbb{N}$  are not equal, but they are **isomorphic**.

```
let in_ : (unit, nat) sum -> nat =  
  function Left () -> Zero | Right n -> Succ n  
let out : nat -> (unit, nat) sum =  
  function Zero -> Left () | Succ n -> Right n
```

Algebraic data types are a form of **nominal iso-recursive types**.

## An alternative syntax

In the usual syntax, the type of lists is declared as follows:

```
type 'a list =  
  | Nil  
  | Cons of 'a * 'a list
```

In the alternative syntax, the type of each data constructor is given:

```
type _ list =  
  | Nil : 'a list  
  | Cons : 'a * 'a list -> 'a list
```

The result type of each constructor is 'a list.

Each constructor is polymorphic in 'a. This is implicit.



## Analogy with inductive types

Coq has **inductive types**, which seem similar to algebraic data types.

It offers similar syntaxes:

```
Inductive list (A : Type) : Type :=  
| Nil  
| Cons (x : A) (xs : list A).
```

```
Inductive list (A : Type) : Type :=  
| Nil : list A  
| Cons : A -> list A -> list A.
```

However, inductive types must be **strictly positive**.

### Data types

Primitive sums,  
products, and  
recursive types

Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

## Analogy with inductive types

Coq has **inductive types**, which seem similar to algebraic data types.

It offers similar syntaxes:

**Inductive List**

**Strict positivity**

The constants  $X_1 \dots X_k$  occur strictly positively in  $T$  in the following cases:

- no  $X_1 \dots X_k$  occur in  $T$
- $T$  converts to  $(X_j t_1 \dots t_q)$  for some  $j$  and no  $X_1 \dots X_k$  occur in any of  $t_i$
- $T$  converts to  $\forall x : U, V$  and  $X_1 \dots X_k$  occur strictly positively in type  $V$  but none of them occur in  $U$
- $T$  converts to  $(I a_1 \dots a_r t_1 \dots t_s)$  where  $I$  is the name of an inductive definition of the form

$\text{Ind } [r] (I : A := c_1 : \forall p_1 : P_1, \dots \forall p_r : P_r, C_1; \dots; c_n : \forall p_1 : P_1, \dots \forall p_r : P_r, C_n)$

(in particular, it is not mutually defined and it has  $r$  parameters) and no  $X_1 \dots X_k$  occur in any of the  $t_i$  nor in any of the  $a_j$  for  $m < j \leq r$  where  $m \leq r$  is the number of recursively uniform parameters, and the (instantiated) types of constructor  $C_i \{p_j/a_j\}_{j=1 \dots m}$  of  $I$  satisfy the nested positivity condition for  $X_1 \dots X_k$

## Analogy with inductive types

Coq has **inductive types**, which seem similar to algebraic data types.

It offers similar syntaxes:

```
Inductive list (A : Type) : Type :=
| Nil
| Cons (x : A) (xs : list A).
```

```
Inductive list (A : Type) : Type :=
| Nil : list A
| Cons : A -> list A -> list A.
```

However, inductive types must be **strictly positive**. In each argument of each constructor, **list** cannot appear in the left of an arrow.

## Algebraic data types are recursive types

### Data types

Primitive sums,  
products, and  
recursive types

Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

Algebraic data types are unrestricted: they are true **recursive** types.

In fact, in Haskell,  $\mu$  itself can be defined as an algebraic data type:

```
data Fix f =           -- the algebraic data type Fix f
    Fix (f (Fix f))    -- has one constructor, also named Fix
```

The parameter **f** has kind  $\star \rightarrow \star$ . It is itself a parameterized type.

If a non-recursive type of lists is defined as follows,

```
data ListF a self = Nil | Cons a self
```

then **Fix** (**ListF** a) is a recursive type of lists.

## Data types

Primitive sums,  
products, and  
recursive types

Algebraic data

Scott & Church

## Existentials

Examples

Metatheory

Church

# 1 Data types

Primitive sums, products, and recursive types

Algebraic data types

Scott and Church encodings

# 2 Existential types

Examples

Metatheory

Church encoding

# Encoding Booleans

The Boolean type  $\mathbb{B} \simeq 1 + 1$  can be declared as an algebraic data type:

```
type bool = False | True
```

However, Booleans can also be **encoded** in pure  $\lambda$ -calculus.

A Boolean value is an “object with a case method”.

It can choose between two branches:

$$\begin{aligned}\mathbb{B} &\triangleq \forall X. (1 \rightarrow X) \rightarrow (1 \rightarrow X) \rightarrow X \\ \text{False} &\triangleq \lambda x_1. \lambda x_2. x_1 () \\ \text{True} &\triangleq \lambda x_1. \lambda x_2. x_2 () \\ \text{case } t \text{ of } t_1 \parallel t_2 &\triangleq t \ t_1 \ t_2\end{aligned}$$

This is a **Scott encoding**, and also a **Church encoding**.

**Exercise:** reconstruct the omitted type abstractions and applications.

## Encoding sums

### Data types

Primitive sums,  
products, and  
recursive types

Algebraic data

Scott & Church

### Existentials

Examples

Metatheory

Church

More generally, the binary sum type  $T_1 + T_2$  can be encoded as follows:

$$\begin{aligned} T_1 + T_2 &\triangleq \forall X. (T_1 \rightarrow X) \rightarrow (T_2 \rightarrow X) \rightarrow X \\ \text{inj}_1 x &\triangleq \lambda x_1. \lambda x_2. x_1 x \\ \text{inj}_2 x &\triangleq \lambda x_1. \lambda x_2. x_2 x \\ \text{case } t \text{ of } t_1 \parallel t_2 &\triangleq t t_1 t_2 \end{aligned}$$

The zero-ary sum type 0 can be encoded, too!

$$\begin{aligned} 0 &\triangleq \forall X. X \\ \text{absurd } t &\triangleq t \end{aligned}$$

Clearly this works for any number of branches.

## Data types

Primitive sums,  
products, and  
recursive types

Algebraic data

Scott & Church

## Existentials

Examples

Metatheory

Church

The binary product type  $T_1 \times T_2$  can be encoded as follows:

$$\begin{aligned} T_1 \times T_2 &\triangleq \forall X. (T_1 \rightarrow T_2 \rightarrow X) \rightarrow X \\ (x_1, x_2) &\triangleq \lambda k. k \ x_1 \ x_2 \\ \pi_1 \ t &\triangleq t \ (\lambda x_1. \lambda x_2. x_1) \\ \pi_2 \ t &\triangleq t \ (\lambda x_1. \lambda x_2. x_2) \end{aligned}$$

The zero-ary product type 1 can be encoded, too!

$$\begin{aligned} 1 &\triangleq \forall X. X \rightarrow X \\ () &\triangleq \lambda x. x \end{aligned}$$

Clearly this works for any number of tuple components.



## Encoding natural integers

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

Can we encode the recursive type  $\mathbb{N} \simeq 1 + \mathbb{N}$  in the same way, à la Scott?

$$\mathbb{N} \triangleq \forall X. (1 \rightarrow X) \rightarrow (\mathbb{N} \rightarrow X) \rightarrow X$$

This doesn't work in System  $F$ , which doesn't have recursive types.

Here, [the Scott and Church encodings differ](#).

The Church encoding views a number as “an object with a *fold* method”.

$$\begin{aligned}\mathbb{N} &\triangleq \forall X. X \rightarrow (X \rightarrow X) \rightarrow X \\ \text{Zero} &\triangleq \lambda z. \lambda s. z \\ \text{Succ } x &\triangleq \lambda z. \lambda s. s (x \ z \ s)\end{aligned}$$

## Encoding lists

The Church encoding views a list as “an object with a *fold* method”.

$$\begin{aligned}\mathbb{L}(Y) &\triangleq \forall X. X \rightarrow (Y \rightarrow X \rightarrow X) \rightarrow X \\ [] &\triangleq \lambda n. \lambda c. n \\ x :: xs &\triangleq \lambda n. \lambda c. c\ x\ (xs\ n\ c)\end{aligned}$$

The Church encoding works for all **inductive types**.

Girard, Taylor, Lafont, **Proofs and types**, 1990, §11.3–11.5.

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

# 1 Data types

Primitive sums, products, and recursive types

Algebraic data types

Scott and Church encodings

# 2 Existential types

Examples

Metatheory

Church encoding

#### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

#### Existentials

##### Examples

Metatheory  
Church

## 1 Data types

Primitive sums, products, and recursive types

Algebraic data types

Scott and Church encodings

## 2 Existential types

Examples

Metatheory

Church encoding

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

Complex numbers are an **abstract concept**.

Outside of their implementation, how they are represented **should be irrelevant**, and one should not depend on implementation details.

*In one section, Professor Descartes announced that a complex number was an ordered pair of reals [...].*

*In the other section, Professor Bessel announced that a complex number was an ordered pair of reals, the first of which was nonnegative [...].*

*An unfortunate mistake [...] caused the two sections to be interchanged.*

Reynolds, **Types, Abstraction and Parametric Polymorphism**, 1983.

## Complex numbers as an abstract type

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

In OCaml, one might implement complex numbers as an **abstract type**:

```
module Complex : sig
  type t
  val zero: t
  val one: t
  val add: t -> t -> t
  val mul: t -> t -> t
  val (=): t -> t -> bool
  (* etc. *)
end
```

## Complex numbers as an existential type

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

In System  $F$ , this idea can be made precise via an **existential type**:

$$\text{Complex} : \exists X. \left\{ \begin{array}{l} \text{zero} : X \\ \text{add} : X \rightarrow X \rightarrow X \\ \text{mul} : X \rightarrow X \rightarrow X \\ \text{eq} : X \rightarrow X \rightarrow \text{bool} \\ \text{etc.} \end{array} \right\}$$

Mitchell and Plotkin, **Abstract types have existential type**, 1988.

Rossberg, Russo, Dreyer, **F-ing Modules**, 2014.

## Streams as an existential type

Imagine we wish to define an abstract type of **streams**.

A stream is a **producer** of a sequence of elements,  
out of which a **consumer** can **pull** elements on demand.

It is an “object” with a single method, *next*.

- a stream has a certain **current internal state**.
- *next* returns either nothing or a pair of an element and a new state.

A stream is analogous to a Java iterator, except it is **not mutable**.  
Its current state is explicit.

$$Stream(X) \simeq \exists S. \underbrace{(S \rightarrow 1 + X \times S)}_{next} \times \underbrace{S}_{cur}$$



## Streams as an existential type

How do we translate this equation in OCaml?

$$\text{Stream}(X) \simeq \exists S. (S \rightarrow 1 + X \times S) \times S$$

We first define **the sum type**  $1 + X \times S$  as an algebraic data type:

$$\text{Step } X \ S \simeq 1 + X \times S$$

so the equation becomes:

$$\text{Stream}(X) \simeq \exists S. (S \rightarrow \text{Step } X \ S) \times S$$

Then we define this **existential type** as an algebraic data type with one data constructor whose type is

$$\forall S. (S \rightarrow \text{Step } X \ S) \times S \rightarrow \text{Stream}(X)$$

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

## Streams as an existential type

`('a, 's)` step corresponds to *Step*  $X$   $S$  and is isomorphic to  $1 + X \times S$ :

```
type ('a, 's) step =  
  | Done                                     (* the stream is exhausted *)  
  | Yield of 'a * 's                       (* here is an element and a new state *)
```

An existential type can be defined as an **algebraic data type**:

```
type 'a stream =  
  | Stream:  
      (* The [next] method: *) ('s -> ('a, 's) step) *  
      (* The current state: *) 's  
      (* together form a stream: *) -> 'a stream
```

The data constructor **Stream** has **universal type**: it is polymorphic in `'s`.

The producer chooses the type of the internal state;  
the consumer must treat this type as abstract.

## Converting a list to a stream

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

This conversion function is a nonrecursive **producer**:

```
let stream (xs : 'a list) : 'a stream =  
  let next xs =  
    match xs with  
    | [] -> Done  
    | x :: xs -> Yield (x, xs)  
  in  
  Stream (next, xs) (* packing an existential type *)
```

On the last line, what is the concrete type of states?

## Converting a list to a stream

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

This conversion function is a nonrecursive **producer**:

```
let stream (xs : 'a list) : 'a stream =  
  let next xs =  
    match xs with  
    | [] -> Done  
    | x :: xs -> Yield (x, xs)  
  in  
  Stream (next, xs) (* packing an existential type *)
```

On the last line, what is the concrete type of states?

It is 'a list.

## Converting a stream to a list

This conversion function is a recursive **consumer**:

```
let unstream (Stream (next, s) : 'a stream) : 'a list =
  let rec unfold s =
    match next s with
    | Done          -> []
    | Yield (x, s) -> x :: unfold s
  in
  unfold s
```

The first line uses **pattern matching** to **unpack** an existential type.

What is the type of `unfold`?

## Converting a stream to a list

This conversion function is a recursive **consumer**:

```
let unstream (Stream (next, s) : 'a stream) : 'a list =
  let rec unfold s =
    match next s with
    | Done          -> []
    | Yield (x, s) -> x :: unfold s
  in
  unfold s
```

The first line uses **pattern matching** to **unpack** an existential type.

What is the type of `unfold`?

It is `s -> 'a list`

where `s` is an abstract type introduced by unpacking at line 1.

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

### Examples

Metatheory  
Church

# Examples of stream producers

How would you implement a singleton stream?

## Examples of stream producers

How would you implement a singleton stream?

```
let return (x : 'a) : 'a stream =
  let next s =
    if s then Yield (x, false) else Done
  in
  Stream (next, true)           (* packing an existential type *)
```

On the last line, the concrete type of states is **bool**:  
either we have already yielded an element, or we have not.

**Exercise:** Write interval of type **int** -> **int** -> **int** stream.

**Exercise:** Write append of type 'a stream -> 'a stream -> 'a stream.



## An example consumer-and-producer

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

The map function on streams is also non-recursive:

```
let map (f : 'a -> 'b) (xs : 'a stream) : 'b stream =  
  let Stream (next, s) = xs in (* unpacking *)  
  let next s =  
    match next s with  
    | Done          -> Done  
    | Yield (x, s) -> Yield (f x, s)  
  in  
  Stream (next, s) (* packing *)
```

## Existential types enforce abstraction

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

When a stream is **unpacked**, a fresh unknown type 's is introduced.

Unpacking two distinct streams gives rise to two **distinct** types:

```
let wrong (xs1 : 'a stream) (xs2 : 'a stream) =
  match xs1, xs2 with
  | Stream (next1, s1), Stream (next2, s2) ->
    next1 s2
```

Error: This expression has type \$Stream\_'s1  
but an expression was expected of type \$Stream\_'s

**Fortunately**, the “next” function of stream 1  
cannot be applied to the internal state of stream 2.

## Streams as an existential type

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

#### Examples

Metatheory  
Church

This encoding of streams is used in practice.

In addition to **Done** and **Yield**, a third constructor **Skip** can be used, meaning “please ask again”

A consumer must ask, ask, ask until a non-**Skip** result is produced.

This allows most stream producers to be **nonrecursive** functions.  
This makes optimization easier.

Coutts, Leshchinskiy, Stewart, **Stream fusion:  
from lists to streams to nothing at all**, 2007.

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples

**Metatheory**

Church

# 1 Data types

Primitive sums, products, and recursive types

Algebraic data types

Scott and Church encodings

# 2 Existential types

Examples

**Metatheory**

Church encoding

System  $F$  with existential types

The syntax of types is extended with **existential types**:

$$T ::= \dots \mid \exists X. T$$

The syntax of terms is extended with **introduction** and **elimination** forms:

$$\begin{aligned} t &::= \dots \mid \text{pack } T, t \text{ as } \exists X. T \mid \text{let } X, x = \text{unpack } t \text{ in } t \\ v &::= \dots \mid \text{pack } T, v \text{ as } \exists X. T \\ E &::= \dots \mid \text{pack } T, E \text{ as } \exists X. T \mid \text{let } X, x = \text{unpack } E \text{ in } t \end{aligned}$$

A new reduction rule is introduced:

$$\text{let } X, x = \text{unpack } (\text{pack } T', v \text{ as } \exists X. T) \text{ in } t \longrightarrow t[v/x][T'/X]$$

Note: “*unpack t*” is not a term. Only “*let ... unpack ... in ...*” is a term.

System  $F$  with existential types

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

Two new typing rules are introduced:

 $\exists$ -INTRO

$$\frac{\Gamma \vdash t :}{\Gamma \vdash \text{pack } T', t \text{ as } \exists X.T : \exists X.T}$$

 $\exists$ -ELIM

$$\frac{\Gamma \vdash t_1 :}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

For reference, recall the typing rules for universal types:

 $\forall$ -INTRO

$$\frac{\Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T}$$

 $\forall$ -ELIM

$$\frac{\Gamma \vdash t : \forall X.T}{\Gamma \vdash t \ T' : T[T'/X]}$$

**Exercise:** extend the proofs of Subject Reduction and Progress.

System  $F$  with existential types

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

Two new typing rules are introduced:

$$\frac{\exists\text{-INTRO} \quad \Gamma \vdash t : T[T'/X]}{\Gamma \vdash \text{pack } T', t \text{ as } \exists X. T : \exists X. T}$$

$$\frac{\exists\text{-ELIM} \quad \Gamma \vdash t_1 : \quad}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

For reference, recall the typing rules for universal types:

$$\frac{\forall\text{-INTRO} \quad \Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X. t : \forall X. T}$$

$$\frac{\forall\text{-ELIM} \quad \Gamma \vdash t : \forall X. T}{\Gamma \vdash t T' : T[T'/X]}$$

**Exercise:** extend the proofs of Subject Reduction and Progress.

System  $F$  with existential types

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

Two new typing rules are introduced:

$$\frac{\exists\text{-INTRO} \quad \Gamma \vdash t : T[T'/X]}{\Gamma \vdash \text{pack } T', t \text{ as } \exists X.T : \exists X.T}$$

$$\frac{\exists\text{-ELIM} \quad \Gamma \vdash t_1 : \exists X.T}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

For reference, recall the typing rules for universal types:

$$\frac{\forall\text{-INTRO} \quad \Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T}$$

$$\frac{\forall\text{-ELIM} \quad \Gamma \vdash t : \forall X.T}{\Gamma \vdash t T' : T[T'/X]}$$

**Exercise:** extend the proofs of Subject Reduction and Progress.



System  $F$  with existential types

## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

Two new typing rules are introduced:

$$\frac{\exists\text{-INTRO} \quad \Gamma \vdash t : T[T'/X]}{\Gamma \vdash \text{pack } T', t \text{ as } \exists X.T : \exists X.T}$$

$$\frac{\exists\text{-ELIM} \quad \begin{array}{c} \Gamma \vdash t_1 : \exists X.T \\ \Gamma; X; x : T \vdash t_2 : T_2 \end{array}}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

For reference, recall the typing rules for universal types:

$$\frac{\forall\text{-INTRO} \quad \Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T}$$

$$\frac{\forall\text{-ELIM} \quad \Gamma \vdash t : \forall X.T}{\Gamma \vdash t T' : T[T'/X]}$$

**Exercise:** extend the proofs of Subject Reduction and Progress.

## System $F$ with existential types

Two new typing rules are introduced:

$$\frac{\exists\text{-INTRO} \quad \Gamma \vdash t : T[T'/X]}{\Gamma \vdash \text{pack } T', t \text{ as } \exists X.T : \exists X.T}$$

$$\frac{\exists\text{-ELIM} \quad \begin{array}{c} \Gamma \vdash t_1 : \exists X.T \quad X \# T_2 \\ \Gamma; X; x : T \vdash t_2 : T_2 \end{array}}{\Gamma \vdash \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2}$$

For reference, recall the typing rules for universal types:

$$\frac{\forall\text{-INTRO} \quad \Gamma; X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T}$$

$$\frac{\forall\text{-ELIM} \quad \Gamma \vdash t : \forall X.T}{\Gamma \vdash t T' : T[T'/X]}$$

**Exercise:** extend the proofs of Subject Reduction and Progress.

#### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

#### Existentials

Examples  
Metatheory  
Church

## 1 Data types

Primitive sums, products, and recursive types

Algebraic data types

Scott and Church encodings

## 2 Existential types

Examples

Metatheory

Church encoding

## Universal/existential duality

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

When a value has universal type  $\forall X.T$ ,  
the **producer** of this value must treat  $X$  as abstract  
and the **consumer** can choose a type  $T'$  with which to instantiate  $X$ .

When a value has existential type  $\exists X.T$ ,  
the **producer** chooses a type  $T'$  with which to instantiate  $X$   
but the **consumer** must treat  $X$  as abstract.

When a value has existential type, **its consumer must be polymorphic**.

## Church encoding of existential types

Existential types can in fact be **encoded** in terms of universal types:

$$\exists X. T \triangleq \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y$$

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

## Church encoding of existential types

Existential types can in fact be **encoded** in terms of universal types:

$$\exists X. T \triangleq \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y$$

As the wizard was studying the black box, suddenly the box spoke:

*I hold a  $T$ , but I cannot give it to you,  
because I cannot reveal  $X$ .*

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

## Church encoding of existential types

Existential types can in fact be **encoded** in terms of universal types:

$$\exists X. T \triangleq \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y$$

As the wizard was studying the black box, suddenly the box spoke:

*I hold a  $T$ , but I cannot give it to you,  
because I cannot reveal  $X$ .*

*What do you want to use it for?*

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

## Church encoding of existential types

Existential types can in fact be **encoded** in terms of universal types:

$$\exists X. T \triangleq \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y$$

As the wizard was studying the black box, suddenly the box spoke:

*I hold a  $T$ , but I cannot give it to you,  
because I cannot reveal  $X$ .*

*What do you want to use it for?*

*Tell me how you wish to transform a  $T$  into a  $Y$ ,  
in a way that works for every  $X$ .  
Then I will give you a  $Y$ .*

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church



## Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

## Existentials

Examples  
Metatheory  
Church

# Church encoding of existential types

$$\exists X. T \triangleq \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y$$

## Church encoding of existential types

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

$$\begin{array}{lcl} \exists X. T & \triangleq & \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y \\ \text{pack } T', v \text{ as } \exists X. T & \triangleq & \end{array}$$

## Church encoding of existential types

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

$$\begin{aligned}
 \exists X. T &\triangleq \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y \\
 \text{pack } T', v \text{ as } \exists X. T &\triangleq \Lambda Y. \lambda k : (\forall X. T \rightarrow Y). k \ T' \ v \\
 \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2 &\triangleq
 \end{aligned}$$

## Church encoding of existential types

### Data types

Primitive sums,  
products, and  
recursive types  
Algebraic data  
Scott & Church

### Existentials

Examples  
Metatheory  
Church

$$\begin{aligned} \exists X. T &\triangleq \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y \\ \text{pack } T', v \text{ as } \exists X. T &\triangleq \Lambda Y. \lambda k : (\forall X. T \rightarrow Y). k \ T' \ v \\ \text{let } X, x = \text{unpack } t_1 \text{ in } t_2 : T_2 &\triangleq t_1 \ T_2 \ (\Lambda X. \lambda x : T \rightarrow T_2. t_2) \end{aligned}$$

This encoding validates the logical implication  $\exists X. T \rightarrow \neg \forall X. \neg T$  where  $\neg T$  is defined as  $T \rightarrow 0$ .

**Exercise:** check that this encoding validates the reduction rule and the typing rules proposed earlier for primitive existential types.