MPRI 2.4

Algebraic data types and existential types

François Pottier

2024

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

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.

Binary products

It is easy to add pairs and projections to the (call-by-value) λ-calculus.

$$\begin{array}{rcl} t & ::= & \ldots \mid (t, t) \mid \pi_i\ t \\ v & ::= & \ldots \mid (v, v) \\ E & ::= & \ldots \mid (E, t) \mid (v, E) \mid \pi_i\ E \end{array} \qquad \text{where } i \in \{1, 2\}$$

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

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

Two new typing rules are needed:

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

Exercise: extend the proofs of Subject Reduction and Progress.

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

Unit

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

It has an introduction form but no elimination form.

$$
\begin{array}{rcl}
t & ::= & \ldots \mid () \\
v & ::= & \ldots \mid ()
\end{array}
$$

– no new evaluation context

No new reduction rule is needed.

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

One new typing rule is needed:

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

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

Binary sums

Let us add injections and a case analysis to (call-by-value) λ-calculus.

$$\begin{array}{lcll} t & ::= & \ldots \mid \mathit{inj}_i\ t \ \mid\ \mathit{case}\ t\ \mathit{of}\ t_1 \parallel t_2 & \qquad \text{where } i \in \{1,2\} \\ v & ::= & \ldots \mid \mathit{inj}_i\ v & \\ E & ::= & \ldots \mid \mathit{inj}_i\ E \ \mid\ \mathit{case}\ E\ \mathit{of}\ t_1 \parallel t_2 & \end{array}$$

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

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

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

Two new typing rules are needed:

$$\frac{\Gamma \vdash t : T_i}{\Gamma \vdash \mathit{inj}_i\ t : T_1 + T_2} \qquad \frac{\begin{array}{c}\Gamma \vdash t : T_1 + T_2 \\ \Gamma \vdash t_1 : T_1 \to T' \qquad \Gamma \vdash t_2 : T_2 \to T'\end{array}}{\Gamma \vdash \mathit{case}\ t\ \mathit{of}\ t_1 \parallel t_2 : T'}$$

Exercise: extend the proofs of Subject Reduction and Progress.

Void

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

It has an elimination form but no introduction form.

$$
\begin{array}{rcl}
t & ::= & \ldots \mid \mathit{absurd}\ t \\
 & & \text{– no new value} \\
E & ::= & \ldots \mid \mathit{absurd}\ E
\end{array}
$$

No new reduction rule is needed. $\mathit{absurd}\ v$ is stuck.

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

One new typing rule is needed:

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

Exercise: extend the proof of Progress.

Approaches to recursive types

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 $\mu X.X \rightarrow X$.

μ 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

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,

- $inj_1\ ()$ 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

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

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

The proof of type soundness is unchanged.

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

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 $\textit{lists} = \mu X.\mathbb{L}(X) = \mathbb{L}(\textit{lists}) = \mathbb{L}(\mathbb{L}(\ldots))$.

```
# 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.

Approach 2: structural iso-recursive types

Suppose we want types to remain finite trees.

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

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

$$
\begin{array}{lcl}
t & ::= & \ldots \mid \mathit{fold}_{\mu X.T}\ t \mid \mathit{unfold}_{\mu X.T}\ t \\
v & ::= & \ldots \mid \mathit{fold}_{\mu X.T}\ v \\
E & ::= & \ldots \mid \mathit{fold}_{\mu X.T}\ E \mid \mathit{unfold}_{\mu X.T}\ E
\end{array}
$$

Their operational semantics is simple:

$$
\mathit{unfold}_{\mu X.T}\ (\mathit{fold}_{\mu X.T}\ v) \quad \longrightarrow \quad v
$$

Two new typing rules are introduced:

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

Approach 2: structural iso-recursive types

$\mathit{fold}_{\mu X.T}$ and $\mathit{unfold}_{\mu X.T}$ 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 λ-calculus with iso-recursive types. Update the proof of type soundness where needed.

Approach 2: structural iso-recursive types

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,

- $inj_1\ ()$ has type $1 + \mathbb{N}$, and also has type $1 + \mathbb{N} \times \mathbb{L}(\mathbb{N})$,
- $fold_{\mathbb{N}}\ (inj_1\ ())$ has type $\mathbb{N}$.
- $fold_{\mathbb{L}(\mathbb{N})}\ (inj_1\ ())$ 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: $T ::= \ldots \mid \mathbb{N}$.

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

$$\frac{\Gamma \vdash t : 1}{\Gamma \vdash \mathit{inj}_1\ t : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathit{inj}_2\ t : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N} \qquad \Gamma \vdash t_1 : 1 \to T' \qquad \Gamma \vdash t_2 : \mathbb{N} \to T'}{\Gamma \vdash \mathit{case}\ t\ \mathit{of}\ t_1 \parallel t_2 : T'}$$

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 $\mathit{in} : 1 + \mathbb{N} \to \mathbb{N}$ and $\mathit{out} : \mathbb{N} \to 1 + \mathbb{N}$ such that $\mathit{in} \cdot \mathit{out} \equiv_{\beta\eta} \mathit{out} \cdot \mathit{in} \equiv_{\beta\eta} \mathit{id}$. This is an iso-recursive approach.

In this approach, there is no μ syntax or μ 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: $T ::= \ldots \mid \mathbb{L}(T)$.

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

$$\frac{\Gamma \vdash t : 1}{\Gamma \vdash inj_1\ t : \mathbb{L}(T)} \qquad \frac{\Gamma \vdash t : T \times \mathbb{L}(T)}{\Gamma \vdash inj_2\ t : \mathbb{L}(T)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \mathbb{L}(T) \\ \Gamma \vdash t_1 : 1 \rightarrow T' \qquad \Gamma \vdash t_2 : T \times \mathbb{L}(T) \rightarrow T'\end{array}}{\Gamma \vdash case\ t\ of\ t_1 \parallel t_2 : T'}$$

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.

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

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

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

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
|
|

In
|
|
```

Ho
eac

Strict positivity

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

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

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

(in particular, it is not mutually defined and it has r parameters) and no $X_1 \ldots X_k$ 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 $C_i\{p_j/a_j\}_{j=1\ldots m}$ of I satisfy the nested positivity condition for $X_1 \ldots X_k$

Algebraic data types are recursive types

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

This breaks strong normalization.

```
type term =
  T of (term -> term) (* not strictly positive! *)

let app (t : term) (u : term) : term =
  match t with T t -> t u

let delta : term =
  T (fun x -> app x x)

let omega : term =
  app delta delta     (* diverges! *)
```

`app delta delta` reduces to itself in one step.

Algebraic data types are recursive types

In Haskell, μ 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 $\star \to \star$. 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.

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 λ-calculus.

A Boolean value is an "object with a *case* method".
It can choose between two branches:

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

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

Exercise: reconstruct the omitted type abstractions and applications.

Encoding sums

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

$$\begin{array}{rcl} T_1 + T_2 & \triangleq & \forall X.\,(T_1 \to X) \to (T_2 \to X) \to X \\ \mathit{inj}_1\ x & \triangleq & \lambda x_1.\lambda x_2.x_1\ x \\ \mathit{inj}_2\ x & \triangleq & \lambda x_1.\lambda x_2.x_2\ x \\ \mathit{case}\ t\ \mathit{of}\ t_1 \parallel t_2 & \triangleq & t\ t_1\ t_2 \end{array}$$

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

$$\begin{array}{rcl} 0 & \triangleq & \forall X.\,X \\ \mathit{absurd}\ t & \triangleq & t \end{array}$$

Clearly this works for any number of branches.

Encoding products

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

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

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

$$\begin{array}{rcl} 1 & \triangleq & \forall X.\,X \to X \\ () & \triangleq & \lambda x.x \end{array}$$

Clearly this works for any number of tuple components.

Encoding natural integers

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

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

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".

$$\begin{array}{rcl} \mathbb{N} & \triangleq & \forall X.\, X \rightarrow (X \rightarrow X) \rightarrow X \\ \mathit{Zero} & \triangleq & \lambda z.\, \lambda s.\, z \\ \mathit{Succ}\, x & \triangleq & \lambda z.\, \lambda s.\, s\, (x\; z\; s) \end{array}$$

Encoding lists

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

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

The Church encoding works for all inductive types.

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

Motivation

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

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

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

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

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.

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

Streams as an existential type

How do we translate this equation in OCaml?

$$Stream(X) \;\simeq\; \exists S. \quad (S \rightarrow 1 + X \times S) \times S$$

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

$$Step\ X\ S \simeq 1 + X \times S$$

so the equation becomes:

$$Stream(X) \;\simeq\; \exists S. \quad (S \rightarrow Step\ X\ S) \times S$$

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

$$\forall S. \quad (S \rightarrow Step\ X\ S) \times S \rightarrow Stream(X)$$

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

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`?

It is `s -> 'a list`
where `s` is an abstract type introduced by unpacking at line 1.

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

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

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

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.

System F with existential types

The syntax of types is extended with existential types:

$$T ::= \ldots \mid \exists X.\, T$$

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

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

A new reduction rule is introduced:

$$\begin{aligned}
\mathit{let}\ X, x = \mathit{unpack}\ (\mathit{pack}\ T', v\ \mathit{as}\ \exists X.T)\ \mathit{in}\ t \quad &\longrightarrow \\
t[v/x][T'/X] &
\end{aligned}$$

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

System F with existential types

Two new typing rules are introduced:

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

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

For reference, recall the typing rules for universal types:

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

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

Exercise: extend the proofs of Subject Reduction and Progress.

Universal/existential duality

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:

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

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.

Church encoding of existential types

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

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.