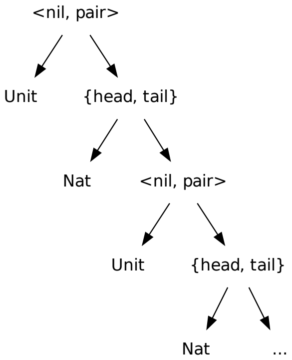
Recursive types

Course on type systems, session #11, 120516.

linus@cs.lth.se

The need for recursive types

- Counter = { get: Counter $\rightarrow$ Nat, inc: Counter $\rightarrow$ Unit }
- ListOfNat = <nil: Unit, pair: { head: Nat, tail: ListOfNat }>
- List T = <nil: Unit, pair: { head: T, tail: List T }>



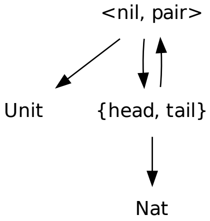
Anonymous recursive types

Just like `fix` produces anonymous recursive functions, we'll use μ to produce anonymous recursive types.

$\text{ListOfNat} = \mu\ \text{X}.\ <\text{nil: Unit, pair: } \{ \text{ head: Nat, tail: X } \}>$

"ListOfNat is defined as the infinite type which satisfies the equation:

$$\text{X} = <\text{nil: Unit, pair: } \{ \text{ head: Nat, tail: X } \}>\text{"}$$



Example code for ListOfNat (1/2)

```
nil = <nil=unit> as ListOfNat;

cons = λh:Nat. λt:ListOfNat.
         <pair={head=h,tail=t}> as ListOfNat;

isnil = λl:ListOfNat. case l of
           <nil=u> ⇒ true
         | <pair=p> ⇒ false;

head = λl:ListOfNat. case l of
           <nil=u> ⇒ 0
         | <pair=p> ⇒ p.head;

tail = λl:ListOfNat. case l of
           <nil=u> ⇒ l
         | <pair=p> ⇒ p.tail;
```

Example code for ListOfNat (2/2)

```
sumlist = fix (λs:ListOfNat→Nat. λl:ListOfNat.
          if isnil l
                    then 0
                    else plus (head l) (s (tail l)));
```

Infinite types – finite values

The type ListOfNat is infinite (circular), but we cannot create infinite data structures due to call-by-value semantics.

What about termination?

Recap

- Remember: STLC took away the ability to express the fixed-point combinator (`fix`), so we had to add it as a primitive.

	Stuck	Terminating	Non-terminating
λ	No	Yes	Yes
λ + extensions	Yes	Yes	Yes
STLC	No	Some	No
STLC + `fix`	No	Yes	Yes

Recap

- Remember: STLC took away the ability to express the fixed-point combinator (`fix`), so we had to add it as a primitive.

	Stuck	Terminating	Non-terminating
λ	No	Yes	Yes
λ + extensions	Yes	Yes	Yes
STLC	No	Some	No
STLC + `fix`	No	Yes	Yes
STLC + μ	No	Yes	Yes

`fix` revisited

fix $=\lambda$f. (λx. f (λy. x x y)) (λx. f (λy. x x y));

`fix` revisited

fix $= \lambda$f. (λx. f (λy. x x y)) (λx. f (λy. x x y));

- `x` must be of an arrow type whose domain is the type of `x` itself: $((\ldots \to X) \to X) \to X$

`fix` revisited

$\text{fix} = \lambda\text{f. } (\lambda\text{x. f } (\lambda\text{y. x x y})) \; (\lambda\text{x. f } (\lambda\text{y. x x y}));$

- `x` must be of an arrow type whose domain is the type of `x` itself: $((\ldots \to X) \to X) \to X$

$T = U{\to}V$

$$\begin{aligned}\text{fix} = \lambda\text{f: } & T{\to}T. \\ & (\lambda\text{x:}(\mu A.\ A{\to}T).\ \text{f } (\lambda\text{y:}(\mu A.\ A{\to}T).\ \text{x x y})) \\ & (\lambda\text{x:}(\mu A.\ A{\to}T).\ \text{f } (\lambda\text{y:}(\mu A.\ A{\to}T).\ \text{x x y}));\end{aligned}$$

$\text{fix} : (T{\to}T) \to T$

Rest eyes here $\longrightarrow$

Hungry functions

Hungry $= \mu$A. Nat $\rightarrow$ A;

Nat $\rightarrow$ Nat $\rightarrow$ Nat $\rightarrow$ Nat $\rightarrow$ ...

- Hungry functions accept any number of arguments.
- Not particularly useful unless we support side-effects:

```
cout << "Hello, " << "world!";
```

vs.

```
hungryPrinter "Hello, " "world!"
```

Objects

- Unlike the objects of last week, these will be immutable.
- Methods may return new objects.
- In the following example, get is a field.

Counter $= \mu$C. {get: Nat, inc: Unit→C};

```
c = let newCounter = fix (
          λf: {x: Nat}→Counter.
          λs: {x: Nat}.
                    {get = s.x,
                     inc = λ_:Unit. f {x=succ(s.x)}})
in newCounter {x=0};
```

An interpreter for untyped λ-calculus

- Every value is a function taking values to values.

D = μX. X$\to$X;

- Explicit folding/unfolding:

lam = λf: D$\to$D. f as D;
ap = λf: D. λa: D. f a;

$$\begin{aligned} \mathit{encode}[\mathtt{x}] &= \mathtt{x} \\ \mathit{encode}[\lambda\mathtt{x}.M] &= \mathtt{lam}(\lambda\mathtt{x} : \mathtt{D}.\ \mathit{encode}[M]) \\ \mathit{encode}[M\ N] &= \mathtt{ap}\ \mathit{encode}[M]\ \mathit{encode}[N] \end{aligned}$$

An interpreter for the extended λ-calculus

The book then extends this interpreter to support the extended λ-calculus ($\lambda + \texttt{Nat}$) in terms of `D`, `lam` and `ap`.

In order for this interpreter to be well-typed, we have to decide what should happen when a `Nat` is applied (as if it were a function) to some value.

```
ap = λf: D. λa: D. case f of
               <nat=n> ⇒ diverge_D unit
             | <fn=f> ⇒ f a;
```

```
diverge_D = λ_:Unit. fix_T (λx: T. x);
```

Hence, stuckness is reintroduced.

What was the point of a type system again? Discuss!

Equivalence (1/2)

Given that

ListOfNat = μ X. <nil: Unit, pair: { head: Nat, tail: X }>

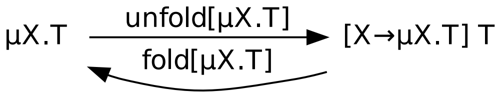
are the following two types the same?

- ListOfNat
- <nil: Unit, pair: { head: Nat, tail: ListOfNat }>

Equivalence (2/2)

Two approaches:

- *Equi-recursive*: Yes, they are the same type. We don't need to change any definitions, safety theorems or proofs.
  - A typechecker for an equi-recursive type system is difficult to implement, because it mustn't get lost in circular data structures. We'll have trouble adding some features to the language (e.g. type operators).
- *Iso-recursive*: No, they are different but *isomorphic*. We add `fold` and `unfold` primitives to the language (sometimes called `roll` and `unroll`).



New rules for the iso-recursive approach (1/3)

t ::= ...
fold[T] t
unfold[T] t

v ::= ...
fold[T] v

T ::= ...
X
μX. T

New rules for the iso-recursive approach (2/3)

$$\text{unfold}[S]\ (\text{fold}[T]\ v_1) \to v_1$$

(E-UnfldFld)

$$\frac{t_1 \to t_1'}{\text{fold}[T]\ t_1 \to \text{fold}[T]\ t_1'}$$

(E-Fld)

$$\frac{t_1 \to t_1'}{\text{unfold}[T]\ t_1 \to \text{unfold}[T]\ t_1'}$$

(E-Unfld)

New rules for the iso-recursive approach (3/3)

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : [X \mapsto U]T_1}{\Gamma \vdash \mathrm{fold}[U]t_1 : U}$$

(T-Fld)

$$\frac{U = \mu X.T_1 \qquad \Gamma \vdash t_1 : U}{\Gamma \vdash \mathrm{unfold}[U]t_1 : [X \mapsto U]T_1}$$

(T-Unfld)

Hiding fold/unfold

In real languages, `fold` and `unfold` are inserted automatically based on the lexical context.

- Insert an implicit `fold` every time a constructor is used.
- Insert an implicit `unfold` in every case statement.

As a consequence of hiding the primitives from the programmer, some code constructs are illegal. For instance, in Haskell, only algebraic datatypes may be recursive – not type aliases.

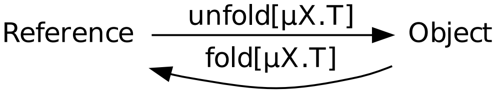
```
data List1 = Nil | Pair Int List1    -- OK

type List2 = (Int, List2)            -- Not allowed
```

Can we still express the fixed-point combinator? Discuss!

Java is iso-recursive

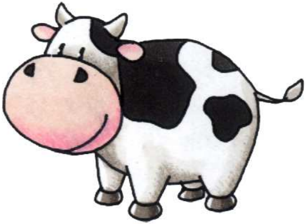
References in Java provide the barrier between the recursive levels.

- Insert an implicit `fold` every time a constructor is used.
- Insert an implicit `unfold` every time we look inside something using the "." operator.



Subtyping – intuition

Suppose we have types for all the integers (Nat) and all the even integers (Even).



Even $<:$ Nat

Now we introduce two function types:

- F $= \mu$X. Nat $\rightarrow$ { value: Even, func: X }
- G $= \mu$X. Even $\rightarrow$ { value: Nat, func: X }

Is F a subtype of G?

$$\text{Remember :} \qquad \frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

Subtyping – equi-recursive approach

For equi-recursive type systems, this is tricky.

The bulk of chapter 21 explains the theoretical foundations of equi-recursive typecheckers.

We will skip it today.

Subtyping – iso-recursive approach

The *Amber rule*, named after the Amber programming language (1986).

$$\frac{\Sigma, X <: Y \vdash S <: T}{\Sigma \vdash \mu X.S <: \mu Y.T}$$

(S-Amber)

$$\frac{(X <: Y) \in \Sigma}{\Sigma \vdash X <: Y}$$

(S-Assumption)

(We also extend the regular subtyping rules to pass along Σ.)

If you think about it long enough, you'll see that it's obvious.

– Saul Gorn