# Computation and Deduction

Lecture 28: Abstract Types
May 3, 2001

1. Intrinsic Formulation
2. Evaluation
3. Examples

# Types

```
tp : type.  %name tp T.

nat   : tp.                          % Natural Numbers
cross : tp -> tp -> tp.              % Pairs
arrow : tp -> tp -> tp.              % Functions
one   : tp.                          % Unit type
plus  : tp -> tp -> tp.              % Sums
mu    : (tp -> tp) -> tp.            % Recursive types
all   : (tp -> tp) -> tp.            % Polymorphic types
exists: (tp -> tp) -> tp.            % Existential types
```

# Expressions I

```
tm : tp -> type.  %name tm E x.

z    : tm nat.
s    : tm nat -> tm nat.
case : tm nat -> tm T -> (tm nat -> tm T) -> tm T.
pair : tm T1 -> tm T2 -> tm (cross T1 T2).
fst  : tm (cross T1 T2) -> tm T1.
snd  : tm (cross T1 T2) -> tm T2.
lam  : (tm T1 -> tm T2) -> tm (arrow T1 T2).
app  : tm (arrow T1 T2) -> tm T1 -> tm T2.
letv : tm T1 -> (tm T1 -> tm T2) -> tm T2.
fix  : (tm T -> tm T) -> tm T.
unit : tm one.
```

# Expressions II

```
inl  : tm T1 -> tm (plus T1 T2).
inr  : tm T2 -> tm (plus T1 T2).
cases: tm (plus T1 T2)
        -> (tm T1 -> tm T) -> (tm T2 -> tm T) -> tm T.
fold : tm (T (mu T)) -> tm (mu T).
unfold : tm (mu T) -> tm (T (mu T)).
tlam : ({a:tp} tm (T a)) -> tm (all T).
tapp : tm (all T) -> {T':tp} tm (T T').
pack : {T':tp} tm (T T') -> tm (exists T).
unpack : tm (exists T1)
          -> ({a:tp} tm (T1 a) -> tm T) -> tm T.
```

# Evaluation I

```
eval : tm T -> tm T -> type.  %name eval D.
%mode eval +E -V.

% Recursive Types
ev_fold : eval (fold E) (fold V)
           <- eval E V.
ev_unfold : eval (unfold E) V
             <- eval E (fold V).

% Polymorphic Types
ev_tlam : eval (tlam E) (tlam E).
ev_tapp : eval (tapp E T) V
           <- eval E (tlam E)
           <- eval (E T) V.
```

# Evaluation II

```
% Existential Types

ev_pack : eval (pack T E) (pack T V)
           <- eval E V.
ev_unpack : eval (unpack E1 E2) V
             <- eval E1 (pack T V1')
             <- eval (E2 T V1') V.

%covers eval' +E *V.
```

# Example: Recursive Types

```
=> = arrow.  %infix right 10 =>.
* = cross.   %infix right 12 *.
+ = plus.    %infix right 11 +.
1 = one.
, = pair.   %infix right 9 ,.
@ = app.    %infix left 10 @.

nats : tp = mu [a] 1 + a.
zero : tm nats = fold (inl unit).
succ : tm (nats => nats) = lam [x] fold (inr x).
```

# Example: Double Function

```
double : tm (nats => nats) =
letv (fold (inl unit) : tm nats) [zero]
letv ((lam [x] fold (inr x)) : tm (nats => nats)) [succ]
fix [d] lam [x]
        cases (unfold x)
        ([x0] zero)
        ([x1] succ @ (succ @ (d @ x1))).
```

# Example: Abstract Types

```
integers =
exists [int]
(nat => int) % toInt
* (int * int => int) % plus
* (int * int => int) % minus
* (int => nat * nat). % fromInt 0-n or n-0
```

# Example: Integers

```
int1 =
pack (nat * nat)
(letv (fix [plus] lam [x] lam [y]
         case x y [x’] s (plus @ x @ y)) [plus]
   letv (fix [norm] lam [x] lam [y]
           case x (x , y)
           [x’] case y (x , y) [y’] norm @ x’ @ y’) [norm]
(lam [n] (n , z)),
 (lam [ij] (plus @ (fst (fst ij)) @ (fst (snd ij)),
               plus @ (snd (fst ij)) @ (snd (snd ij)))),
 (lam [ij] (plus @ (fst (fst ij)) @ (snd (snd ij)),
               plus @ (fst (snd ij)) @ (snd (fst ij)))),
 (lam [i] (norm @ (fst i) @ (snd i))))
: tm integers.
```

# Example: Subtraction

```
test2 =
unpack int1
[int] [ix]
letv (fst ix) [toInt]
letv (fst (snd ix)) [add]
letv (fst (snd (snd ix))) [subtract]
letv (snd (snd (snd ix))) [fromInt]
fromInt @ (subtract @ (toInt @ (s z), toInt @ s (s (s z)))).

%query 1 *
eval test2 V.
```