Functions Representable in $\lambda\!\rightarrow$:

Extended Polynomials

functions generated by addition, multiplication, test for zero

predecessor and subtraction not representable

For more interesting programs we need more expressive types

System T (invented by Kurt Gödel to prove consistency of Arithmetics)

Simply Typed λ-calculus + a special type for natural numbers

$\lambda\!\rightarrow$ + Nat

Types: $T ::= \mathrm{Nat} \mid T \rightarrow T$

↑ instead of a dummy type o a base type of naturals

Rules: Same rules as $\lambda\!\rightarrow$ plus special rules for Nat

$$\frac{}{0 : \mathrm{Nat}} \qquad \frac{n : \mathrm{Nat}}{(\mathrm{S}\ n) : \mathrm{Nat}}$$

] Introduction Rules

$$\frac{h : \mathrm{Nat} \rightarrow T \rightarrow T \quad a : T \quad n : \mathrm{Nat}}{\mathrm{rec}_T\ h\ a\ n : T}$$

Elimination Rule

T can be any type

(Actually: one rule for each type)

Reduction Rules (ι-reduction)

$$\mathrm{rec}_T\ h\ a\ 0 \rightsquigarrow a$$

$$\mathrm{rec}_T\ h\ a\ (S\ n) \rightsquigarrow h\ n\ (\mathrm{rec}_T\ h\ a\ n)$$

General form of rules for a new type

Introduction
How we construct elements of the type

Elimination
How we define functions on the type (recursion principle)

Reduction
How we compute those functions

Transforming a recursive definition with equations to a system T term:

$$\mathrm{plus} : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}$$

$$\mathrm{plus}\ n\ 0 = n$$

$$\mathrm{plus}\ n\ (S\,m) = S\ (\mathrm{plus}\ n\ m)$$

$$\mathrm{plus} := \lambda n{:}\mathrm{Nat}.\ \lambda m{:}\mathrm{Nat}.\ \mathrm{rec}_{\mathrm{Nat}}\ (\lambda m{:}\mathrm{Nat}.\ \lambda k{:}\mathrm{Nat}.\ S\,k)\ n\ m$$

Verify that it works by trying some computations

$$\mathrm{plus}\ \overline{3}\ \overline{2} \rightsquigarrow^* \overline{6}$$

$$\uparrow$$

$$S\ (S\ (S\ 0))$$

Multiplication and Exponentiation:

$$mult : Nat \to Nat \to Nat$$

$$mult = \lambda n{:}Nat.\ \lambda m{:}Nat.\ rec_{Nat}\ (\lambda h{:}Nat.\ \lambda k{:}Nat.\ plus\ k\ n)\ 0\ m$$

$$exp : Nat \to Nat \to Nat$$

$$exp = \lambda n{:}Nat.\ \lambda m{:}Nat.\ rec_{Nat}\ (\lambda h{:}Nat.\ \lambda k{:}Nat.\ mult\ k\ n)\ \bar{1}\ m$$

From now on, we leave out the type of abstracted variables (the can be derived)

Factorial:

$$fact : Nat \to Nat$$

$$fact\ 0 = 1$$

$$fact\ (S\ n) = (S\ n) \cdot (fact\ n)$$

$$fact = \lambda n.\ rec\ (\lambda m.\ \lambda k.\ mult\ k\ (S\ m))\ \bar{1}\ n$$

We can easily define predecessor:

$$pred : Nat \to Nat$$

$$pred = \lambda n.\ rec\ (\lambda m.\ \lambda k.\ m)\ 0\ n$$

Exercise:

- Define subtraction
- What about Fibonacci Numbers

The recursor rec_T can be used with higher types: T can be a function type: **higher-order recursion**

Ackermann Function

$Ack : Nat \to Nat \to Nat$

$Ack\ 0\ m = m+1$

$Ack\ (n+1)\ 0 = Ack\ n\ 1$

$Ack\ (n+1)\ (m+1) =$

$Ack\ n\ (Ack\ (n+1)\ m)$

two recursive calls

nested: one inside the other

In system T:

$$\lambda n.\ rec_{Nat \to Nat}$$
$$(\lambda x.\ \lambda f.$$
$$\lambda m.\ rec_{Nat}$$
$$(\lambda y.\ \lambda k.\ f\ k)$$
$$(f\ \bar{1})\ m)$$
$$(\lambda m.\ S\ m)$$
$$n$$

Trick for readability: give variables suggestive names:

$$Ack =$$
$$\lambda n'.\ rec\ (\lambda n.\ \lambda ackn.$$
$$\lambda m'.\ rec\ (\lambda m.\ \lambda ackSnm.$$
$$ackn\ ackSnm)$$
$$(ackn\ \bar{1})\ m')$$
$$(\lambda m'.\ S\ m')\ n$$