Mugda - Abhängige Typen und Termination-Checking

Karl Mehltretter

1. Februar 2008

Dependent Types

- Martin-Löf Typentheorie
- Beweissysteme wie Coq, Lego (*proofs as programs*).
- Programmiersprachen: Agda, Epigram.

```
conc :: (m n : Nat) -> Lst m -> Lst n -> Lst m + n
```

Termination-Checking

```
length :: Lst -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

- syntaktisches Kriterium
- Verbesserung: z.B. *size-change principle* (Jones et al.) verwenden

- *sized types* (Abel, Barthe, Pareto . . .): Information über Größe im Typsystem mitführen.
- `xs` hat Typ `[a] i`
- `x:xs` hat Typ `[a] (i ++)`

Diplomarbeit

Agda 2 (U. Norrel)

- *inductive families* (P. Dybjer)
- *dependent pattern matching* (Th. Coquand)

. . . braucht einen Termination-Checker.

- Programme (Beweise) sollen total sein.
- entscheidbares Type-Checking.

Mugda

- weniger features . . .
- *sized types*
- coinduktive Typen (z.B. unendliche Listen), Produktivität.

Mugda Beispiel-Programme

```
data Bool : Set
  tt  : Bool
  ff  : Bool


data Nat  : Set
  zero  : Nat
  succ  : Nat  → Nat

fun add  : Nat  → Nat  → Nat
  add x zero  = x
  add x (succ y) = succ (add x y)
```

Listen und Vektoren

$$\begin{array}{l}\mathsf{data\ List}\ (A : \mathsf{Set}) : \mathsf{Set} \\ \quad \mathsf{nil}\ \ : \mathsf{List}\ A \\ \quad \mathsf{cons}\ \ : A \to \mathsf{List}\ A \to \mathsf{List}\ A\end{array}$$

$$\begin{array}{l}\mathsf{data\ Vec}\ (A : \mathsf{Set}) : \mathsf{Nat}\ \to \mathsf{Set} \\ \quad \mathsf{nil}\ \ : \mathsf{Vec}\ A\ \mathsf{zero} \\ \quad \mathsf{cons}\ \ : (n : \mathsf{Nat}\ ) \to A \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (\mathsf{succ}\ n)\end{array}$$

$$\begin{array}{l}\mathsf{fun\ head}\ \ : (A : \mathsf{Set}) \to (n : \mathsf{Nat}\ ) \to \mathsf{Vec}\ A\ (\mathsf{succ}\ n) \to A \\ \quad \mathsf{head}\ \underline{B}\ \underline{m}\ (\mathsf{cons}\ \ B\ m\ x\ xl) = x\end{array}$$

inaccessible pattern

Stream : coinduktiver Datentyp

codata Stream : Set
  cons : Nat → Stream → Stream

cofun zeroes : Stream
  zeroes = cons zero zeroes

fun head : Stream → Nat
  head (cons $x\ xs$) = x

cofun unp : Stream
  unp = unp

cofun unp′ : Stream
  unp′ = cons zero (tail unp′)

Syntax

für Typen und Terme:

$$
\begin{array}{lllll}
\text{EXPR} \ni & e, A, B & ::= & \lambda\, x.\, e \\
& & \mid & (x : A) \to B \\
& & \mid & e\ e_1 \ldots e_n \\
& & \mid & \mathsf{let}\ x : e_1 = A_1\ \mathsf{in}\ e_2 \\
& & \mid & \mathsf{Set} \\
& & \mid & x \mid \mathsf{c} \mid \mathsf{D} \mid \mathsf{f} \mid \mathsf{I}
\end{array}
$$

Patterns:

$$
\begin{array}{lllll}
\text{PAT} \ni & p & ::= & x \\
& & \mid & \mathsf{c}\ \vec{p} \\
& & \mid & \underline{e}
\end{array}
$$

...

Problem: Terminierung

$$\mathsf{fun\ foo\ } : \mathsf{Nat} \rightarrow \mathsf{Nat}$$
$$\mathsf{foo}\ x = \mathsf{foo}\ x$$

$$\mathsf{let}\ v : \mathsf{Vec\ Nat\ (foo\ zero\ )} = \mathsf{nil\ Nat\ zero}$$

- Strukturelle Ordnung:
- $x < \mathsf{c} \ldots\ x\ \ldots$ wenn c induktiver Konstruktor
- D. Wahlstedt 2007: size-change principle + dependent types

Problem: Liste umdrehen ...

```
mutual
  fun rev  : (A : Set) → List A → List A
    rev A (nil A) = nil A
    rev A (cons A x xs) = cons A (rev₁ A x xs) (rev₂ A x xs)

  fun rev₁  : (A : Set) → A → List A → A
    rev₁ A a (nil A) = a
    rev₁ A a (cons A x xs) = rev₁ A x xs

  fun rev₂  : (A : Set) → A → List A → List A
    rev₂ A a (nil A) = nil A
    rev₂ A a (cons A x xs) = rev A (cons A a (rev A (rev₂ A x xs)))
```

... aber mit Vektoren klappts

```
mutual
  fun rev  : (n : Nat) → (A : Set) → Vec A n → Vec A n
    rev zero  A (nil A) = nil A
    rev succ n A (cons A n x xs) =
       cons A n (rev₁ n A x xs) (rev₂ n A x xs)

  fun rev₁  : (n : Nat) → (A : Set) → A → Vec A n → A
    rev₁ zero  A a (nil A) = a
    rev₁ succ n A a (cons A n x xs) = rev₁ n A x xs

  fun rev₂  : (n : Nat) → (A : Set) → A → Vec A n → Vec A n
    rev₂ zero  A a (nil A) = nil A
    rev₂ succ n A a (cons A n x xs) =
       rev (succ n) A (cons A n a (rev  n A (rev₂  n A x xs))
```

Sized types

- Genauere Typen wie Vec statt List helfen bei Terminierung
- Filtern von Vektoren : Existentielle Typen

$$\begin{aligned}\mathsf{fun\ filter} : (A : \mathsf{Set}) \to (p : A \to \mathsf{Bool}) \to (n : \mathsf{Nat}) \\ \to \mathsf{Vec}\ A\ n \to (m : \mathsf{Nat}\ .(\ \mathsf{Leq}\ m\ n\ ,\ \mathsf{Vec}\ A\ m))\end{aligned}$$

- oft reicht upper bound. Automatisches Subtyping für Size:

$$\begin{aligned}\mathsf{fun\ filter} : (A : \mathsf{Set}) \to (p : A \to \mathsf{Bool}) \to (i : \mathsf{Size}) \\ \to \mathsf{List}\ A\ i \to \mathsf{List}\ A\ i\end{aligned}$$

- Nat reicht nicht aus für ein Objekt vom Typ Stream

Füge den speziellen Typ Size zu Mugda hinzu!

Erweiterung von Mugda

- $$\begin{array}{lcll} \text{EXPR} \ni e, A, B & ::= & \ldots & \\ & | & \mathsf{Size} & \text{Size Typ} \\ & | & \mathsf{s}\, e & \text{Nachfolger} \\ & | & \infty & \text{Limit} \\ \text{PAT} \ni p & ::= & \ldots & \\ & | & \mathsf{s}\, p & \end{array}$$

  ...

- $\mathsf{s}\, \infty = \infty$
- für Terminierung: $i < (\mathsf{s}\, i)$ (Admissibility !)

Sized data types

$$\begin{array}{l}
\mathsf{sized\ data\ List}\ (A : \mathsf{Set}) : \mathsf{Size} \to \mathsf{Set} \\
\quad \mathsf{nil}\ \ : (i : \mathsf{Size}) \to \mathsf{List}\ A\ (\mathsf{s}\ i) \\
\quad \mathsf{cons}\ \ : (i : \mathsf{Size}) \to A \to \mathsf{List}\ A\ i \to \mathsf{List}\ A\ (\mathsf{s}\ i)
\end{array}$$

$$\begin{array}{l}
\mathsf{sized\ data\ Nat}\ \ : \mathsf{Size} \to \mathsf{Set} \\
\quad \mathsf{zero}\ \ : (i : \mathsf{Size}) \to \mathsf{Nat}\ \ (\mathsf{s}\ i) \\
\quad \mathsf{succ}\ \ : (i : \mathsf{Size}) \to \mathsf{Nat}\ \ i \to \mathsf{Nat}\ \ (\mathsf{s}\ i)
\end{array}$$

$$\begin{array}{l}
\mathsf{sized\ codata\ Stream}\ \ : \mathsf{Size} \to \mathsf{Set} \\
\quad \mathsf{cons}\ \ : (i : \mathsf{Size}) \to \mathsf{Nat}\ \ \to \mathsf{Stream}\ i\ \ \to \mathsf{Stream}\ (\mathsf{s}\ i)
\end{array}$$

$$\ldots$$

Subtyping

- $\mathsf{List}\ i$ ist Subtyp von $\mathsf{List}\ (\mathsf{s}\ i)$ und $\mathsf{List}\ \infty$.
- $\mathsf{Stream}\ (\mathsf{s}\ i)$ und $\mathsf{Stream}\ \infty$ sind Subtypen von $\mathsf{Stream}\ i$.
- ersetze Gleichheit von Typen durch Subtyping.

Beispiel: Division

$$\begin{aligned}
&\text{fun minus} : (i : \text{Size}) \to \text{Nat}\ i \to \text{Nat}\ \infty \to \text{Nat}\ i \\
&\quad \text{minus}\ \underline{\text{s}\ i}\ (\text{zero}\ i)\ y = \text{zero}\ i \\
&\quad \text{minus}\ i\ x\ (\text{zero}\ \underline{\infty}) = x \\
&\quad \text{minus}\ \underline{\text{s}\ i}\ (\text{succ}\ \ i\ x)\ (\text{succ}\ \underline{\infty}\ y) = \text{minus}\ i\ x\ y \\
\\
&\text{fun div} : (i : \text{Size}) \to \text{Nat}\ i \to \text{Nat}\ \infty \to \text{Nat}\ i \\
&\quad \text{div}\ \underline{\text{s}\ i}\ (\text{zero}\ i)\ y = \text{zero}\ i \\
&\quad \text{div}\ \underline{\text{s}\ i}\ (\text{succ}\ i\ x)\ (\text{zero}\ \underline{\infty}) = \text{zero}\ i \\
&\quad \text{div}\ \underline{\text{s}\ i}\ (\text{succ}\ i\ x)\ (\text{succ}\ \underline{\infty}\ y) = \\
&\qquad \text{succ}\ i\ (\text{div}\ i\ (\text{minus}\ i\ x\ y)\ (\text{succ}\ \infty\ y))
\end{aligned}$$

Streams

$$\begin{aligned}&\mathsf{cofun\ zeroes}\ : (i : \mathsf{Size}) \to \mathsf{Stream}\ i\\ &\quad \mathsf{zeroes}\ (\mathsf{s}\ i) = \mathsf{cons}\ i\ \mathsf{zero}\ (\mathsf{zeroes}\ i)\end{aligned}$$

$$\begin{aligned}&\mathsf{cofun\ unp}\ : (i : \mathsf{Size}) \to \mathsf{Stream}\ i\\ &\quad \mathsf{unp}\ i = \mathsf{unp}\ i\end{aligned}$$

Produktivität über Size statt syntaktischem *Guardedness*-Test.

Admissibility - Nicht zulässiger Typ

bei rekursiven Funktionen:

```
mutual
 fun loop  : (i : Size) → Nat i
           → (Nat ∞ → Maybe (Nat i)) → Bool
   loop s i (zero i) f = loop_case (s i) (zero i) f (f (zero i))
   loop s i (succ i n) f = loop i n (shift i f)

 fun loop_case  : (i : Size) → (Nat ∞ → Maybe (Nat i))
      ...

let inc : Nat ∞ → Maybe Nat ∞ = λ n. just Nat ∞ (succ ∞ n)
let diverge : Bool = loop ∞ (zero ∞) inc
```

Admissibility

bei rekursiven Funktionen:

$$\mathsf{fun}\ \mathsf{bad}_1\ : (i : \mathsf{Size}) \to \mathsf{Bool}$$
$$\mathsf{bad}_1\ (\mathsf{s}\ i) = \mathsf{bad}_1\ i$$

$$\mathsf{fun}\ \mathsf{bad}_2\ : (i : \mathsf{Size}) \to \mathsf{Nat}\ i \to \mathsf{Bool}$$
$$\mathsf{bad}_2\ \underline{\mathsf{s}\ \mathsf{s}\ i}\ (\mathsf{zero}\ (\mathsf{s}\ i)) = \mathsf{bad}_2\ (\mathsf{s}\ i)\ (\mathsf{zero}\ i)$$
$$\mathsf{bad}_2\ \underline{\mathsf{s}\ i}\ (\mathsf{succ}\ i\ x) = \mathsf{bad}_2\ i\ x;$$

Unvollständiges Size pattern ! $(\mathsf{s}\ p)$ darf nicht vorkommen.

Zusamenfassung

- Mugda verbindet *size-change principle* und *sized types*.
- Produktivität mittels *sized types* statt *guardedness*.
- wurde implementiert (mit Type-Checking etc.)

Todo:

- Klauseln sollen alle Fälle abdecken.
- Alles beweisen . . .
- Verbessere Admissiblity, Higher-Order Subtyping . . .

Backup

Gleichheit

$$\text{data Eq } (A : \text{Set}) : A \to A \to \text{Set}$$
$$\quad \text{refl} \ : (a : A) \to \text{Eq } A\ a\ a$$

$$\text{let proof} \ : (x : \text{Nat}\ ) \to \text{Eq Nat (add } x \text{ zero)}\ x$$
$$\qquad = \lambda\, y.\ \text{refl Nat}\ y$$

Listen, Bäume

data List (+ A : Set) : Set
  nil : List A
  cons : $A \to$ List $A \to$ List A

data Tree (+ A : Set) : Set
  node : $A \to$ List (Tree A) $\to$ Tree A

Syntax

$$
\begin{array}{rrl}
\text{DECL} \ni \delta & ::= & \mathsf{data}\ \mathsf{D}\ \tau : A\ \vec{\gamma} \\
& | & \mathsf{codata}\ \mathsf{D}\ \tau : A\ \vec{\gamma} \\
& | & \mathsf{let}\ \mathsf{l} : A\ = e \\
& | & \mathsf{mutual}\ \vec{\mu} \\
& | & \mathsf{mutual}\ \vec{\nu}
\end{array}
$$

$$
\begin{array}{rrl}
\tau & ::= & \diamond \\
& | & (x : A)\ \tau \\
& | & (+\, x : A)\ \tau
\end{array}
$$

$$
\begin{array}{rrl}
\text{CLAUSE} \ni \kappa & ::= & \mathsf{f}\ \vec{p} = e \\
\gamma & ::= & \mathsf{c}\ :\ A \\
\mu & ::= & \mathsf{fun}\ \mathsf{f} : A\ \vec{\kappa} \\
\nu & ::= & \mathsf{cofun}\ \mathsf{f} : A\ \vec{\kappa}
\end{array}
$$

Semantik - Values, Closures

$$
\begin{array}{lll}
\mathrm{VAL} \ni v & ::= & v\,\vec{v} \\
 & | & \mathsf{Lam}\; x\; e^{\rho} \\
 & | & \mathsf{Pi}\; x\; v\; e^{\rho} \\
 & | & a \\
 & & \\
\mathrm{AVAL} \ni a & ::= & k \in \mathbb{N} \\
 & | & \mathsf{Set} \\
 & | & \mathsf{c} \mid \mathsf{f} \mid \mathsf{D} \\
 & & \\
\mathrm{ENV} \ni \rho & ::= & \diamond \\
 & | & \rho, x = v
\end{array}
$$

Auswertung

$$
\begin{aligned}
&\searrow : \mathrm{EXPR} \times \mathrm{ENV} \to \mathrm{VAL} \\
&\searrow (\lambda\, x.\, e)^{\rho} && = \mathsf{Lam}\; x\; e^{\rho} \\
&\searrow ((x : A) \to B)^{\rho} && = \mathsf{Pi}\; x\; v_A\; B^{\rho} \text{ mit } v_A = \searrow A^{\rho} \\
&\searrow (\mathsf{let}\; x : A = e_1\; \mathsf{in}\; e_2)^{\rho} && = \searrow e_2^{\rho, x = v_1} \text{ mit } v_1 = \searrow e_1^{\rho} \\
&\searrow (e\; e_1 \ldots e_n)^{\rho} && = \mathrm{app}\; v\; v_1 \ldots v_n \text{ mit } v = \searrow e^{\rho}, v_i = \searrow e_i^{\rho} \\
&\searrow \mathsf{Set}^{\rho} && = \mathsf{Set} \\
&\searrow \mathsf{c}^{\rho} && = \mathsf{c} \\
&\searrow \mathsf{f}^{\rho} && = \mathsf{f} \\
&\searrow \mathsf{I}^{\rho} && = \searrow e^{\diamond} \text{ mit } \Sigma\, \mathsf{I} = (e, v_t) \\
&\searrow x^{\rho} && = \mathrm{lkup}\; \rho\; x
\end{aligned}
$$

$$\mathrm{app} : \mathrm{VAL} \times \mathrm{VAL}^* \to \mathrm{VAL}$$

$\ldots$

Bidirektionales Type-Checking

Ist e vom Typ A ?

- ist Typ:

$$k; \rho_1; \rho_2 \vdash A \ \mathbf{Type} \subseteq \mathbb{N} \times \mathrm{ENV} \times \mathrm{ENV} \times \mathrm{EXPR}$$

- hat Typ:

$$k; \rho_1; \rho_2 \vdash e \Leftarrow v \subseteq \mathbb{N} \times \mathrm{ENV} \times \mathrm{ENV} \times \mathrm{EXPR} \times \mathrm{VAL}$$

- inferiere Typ:

$$k; \rho_1; \rho_2 \vdash e \Rightarrow v : \mathbb{N} \times \mathrm{ENV} \times \mathrm{ENV} \times \mathrm{EXPR} \to \mathrm{VAL}$$

Gleichheit von Values

- Modus-Wechsel

$$\textsc{chk-inf}\ \frac{k;\rho_1;\rho_2 \vdash e \rightrightarrows v_2 \qquad \mathbf{N};k \vdash v_2 \Leftrightarrow v_1}{k;\rho_1;\rho_2 \vdash e \Leftarrow v_1}$$

- Gleichheit (mit begrenztem Ausrollen von corekursiven Definitionen)

$$\mathbb{F} ::= \{\mathbf{L}, \mathbf{R}, \mathbf{N}\}$$

$$f;k \vdash v_1 \Leftrightarrow v_2 \subseteq \mathbb{F} \times \mathbb{N} \times \mathrm{VAL} \times \mathrm{VAL}$$

$$f;k \vdash v_1 \leftrightarrow v_2 \subseteq \mathbb{F} \times \mathbb{N} \times \mathrm{VAL} \times \mathrm{VAL}$$

- Bisimilarität als codata Prädikat definierbar.
- *observational equality* (McBride)

Type-Checking von Deklarationen

- Datentyp-Deklaration: strikte Positivität etc.
- Funktions-Deklaration : jede Klausel einzeln checken etc.

Terminierung

- ORDER = $\{ ?, \leq, < \}$
- $x <$ c $\ldots$ x $\ldots$ wenn c induktiver Konstruktor
- fun add : Nat $\rightarrow$ Nat $\rightarrow$ Nat
  add x zero $= x$
  add x (succ y) = succ (add x y)
- $\text{add} \begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix} \text{add} \star \text{add} \begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix} \text{add} = \text{add} \begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix} \text{add}$
- size-change principle: Jede idempotente Matrix muss ein $<$ auf Diagonale haben.
- D. Wahlstedt 2007: size-change principle + dependent types

Erweiterung von Mugda

- $$\begin{array}{lcll} \text{EXPR} \ni e, A, B & ::= & \ldots & \\ & | & \mathsf{Size} & \text{Size Typ} \\ & | & \mathsf{s}\, e & \text{Nachfolger} \\ & | & \infty & \text{Limit} \\ \text{PAT} \ni p & ::= & \ldots & \\ & | & \mathsf{s}\, p & \\ \text{VAL} \ni v & ::= & \ldots & \\ & | & \mathsf{Size} & \\ & | & \mathsf{s}\, v & \\ & | & \infty & \end{array}$$
- $\mathsf{s}\, \infty = \infty$
- Terminierung: $i < (\mathsf{s}\, i)$ (Admissibility !)

Subtyping

- List i ist Subtyp von List (s i) und List ∞.
- Stream (s i) und Stream ∞ sind Subtypen von Stream i.
- ersetze Regel:

$$\textsc{chk-inf}\ \frac{k;\rho_1;\rho_2 \vdash e \rightrightarrows v_2 \qquad \mathbf{N};k;\vdash v_2 \leqq v_1}{k;\rho_1;\rho_2 \vdash e \Leftarrow v_1}$$

Streams

$$\text{cofun zeroes} : (i : \text{Size}) \to \text{Stream}\ i$$
$$\text{zeroes}\ (\text{s}\ i) = \text{cons}\ i\ \text{zero}\ (\text{zeroes}\ i)$$

$$\text{cofun unp} : (i : \text{Size}) \to \text{Stream}\ i$$
$$\text{unp}\ i = \text{unp}\ i$$

$$\text{cofun fib}' : \text{Nat} \to \text{Nat} \to (i : \text{Size}) \to \text{Stream}\ i$$
$$\text{fib}'\ x\ y\ (\text{s}\ i) = \text{cons}\ i\ x\ (\text{fib}'\ y\ (\text{add}\ x\ y)\ i)$$
$$\text{let fib} : \text{Stream}\ \infty = \text{fib}'\ (\text{succ zero})\ (\text{succ zero})\ \infty$$

$$\text{fun head} : \text{Stream}\ \infty \to \text{Nat}$$
$$\text{head}\ (\text{cons}\ \underline{\infty}\ x\ xs) = x$$
$$\text{fun tail} : \text{Stream}\ \infty \to \text{Stream}\ \infty$$
$$\text{tail}\ (\text{cons}\ \underline{\infty}\ x\ xs) = xs$$

Admissibility - fun

$$\mathsf{fun\ loop}\ : (i : \mathsf{Size}) \to \mathsf{Nat}\ i \to (\mathsf{Nat}\ \infty \to \mathsf{Maybe}\ (\mathsf{Nat}\ i)) \to \mathsf{Bool}$$

fun f mit Typ $t = (a_1 : A_1) \to \ldots \to (a_n : A_n) \to R$.

Für jedes $(a_j : A_j)$ der Form $(i : \mathsf{Size})$:

- Für $k > j$, entweder i kommt nicht in A_k vor oder A_k ist ein *sized inductive type* der Größe i.
- Der Ergebniss-Typ R ist *monoton* in i.

Ausserdem: Ein Pattern der Form $\mathsf{s}\,p$ kommt nicht in den Klauseln vor.

. . . dann ist f *admissible*.

Admissibility - cofun

cofun zeroes : $(i : \mathsf{Size}) \to \mathsf{Stream}\ i$
zeroes $(\mathsf{s}\ i) = \mathsf{cons}\ i\ \mathsf{zero}\ (\mathsf{zeroes}\ i)$

cofun f mit Typ $t = (a_1 : A_1) \to \ldots \to (a_n : A_n) \to R$.

Für jedes $(a_j : A_j)$ der Form $(i : \mathsf{Size})$:

- Fuer $k > j$, i kommt nicht in A_k vor.
- Der Ergebniss-Typ R ist ein ist ein *sized coinductive type* der Größe i.

. . . dann ist f *admissible*.

Admissibility - cofun

- Folgende Haskell-Definition:

```
fib :: [Int]
fib = (1 : ( 1 : zipWith (+) fib (tail fib)))
```

- Nicht produktiv in Mugda.
- fun fib : $(i : \mathsf{Size}) \to \mathsf{Stream}\ i$
  fib (s s i) = cons (s i) 1 (cons i 1
  (zipWith add (fib i) (tail i (fib (s i)))))
- aber nicht admissble:
  tail : $(i : \mathsf{Size}) \to \mathsf{Stream}\ (\mathsf{s}\ i) \to \mathsf{Stream}\ i$

Implementierung

```
sized data Nat : Size -> Set
{
  zero : (i: Size ) -> Nat ($ i);
  succ : (i : Size ) -> Nat i -> Nat ($ i)
}

fun addWith : ((k : Size ) -> Nat k -> Nat k )
  -> (i : Size ) -> (j : Size )
  -> Nat i -> Nat j -> Nat #
{
  addWith f .($ i) j (zero i) y = y;
  addWith f .($ i) j (succ i x) y =
     succ # (addWith f j i (f j y) (f i x))
}
```