$$\frac{}{Nil \in A^*} \qquad \frac{x \in A \quad \ell \in A^*}{x :: \ell \in A^*}$$

Iteratively defined functions on finite lists

$A^* \stackrel{\text{def}}{=}$ finite lists of elements of the set A

Given a set A', an element $x' \in A'$, and a function $f : A \to A' \to A'$, the *iteratively defined function* $listIter\ x'\ f$ is the unique function $g : A^* \to A'$ satisfying:

$$g\ Nil = x'$$

$$g\ (x :: \ell) = f\ x\ (g\ \ell).$$

for all $x \in A$ and $\ell \in A^*$.

Iteratively defined functions on finite lists

$A^* \stackrel{\text{def}}{=}$ finite lists of elements of the set A

Given a set A', an element $x' \in A'$, and a function $f : A \longrightarrow A' \longrightarrow A'$, the *iteratively defined function* $listIter\, x'\, f$ is the unique function $g : A^* \longrightarrow A'$ satisfying:

$$g\, Nil = x'$$
$$g\,(x :: \ell) = f\, x\,(g\,\ell).$$

for all $x \in A$ and $\ell \in A^*$.

$g\, Nil = x'$

$g(x_1 :: Nil) = f\, x_1\, x'$

$g(x_2 :: x_1 :: Nil) = f x_2 (f x_1\, x')$

$\vdots$

$g(x_n :: \cdots :: x_1 :: nil) = f x_n (\cdots (f x_1\, x') \cdots)$

Iteratively defined functions on finite lists

$A^* \stackrel{\text{def}}{=}$ finite lists of elements of the set A

Given a set A', an element $x' \in A'$, and a function $f : A \to A' \to A'$, the *iteratively defined function* $listIter\, x'\, f$ is the unique function $g : A^* \to A'$ satisfying:

$$g\, Nil = x'$$
$$g\,(x :: \ell) = f\, x\,(g\, \ell).$$

for all $x \in A$ and $\ell \in A^*$.

For each $\ell \in A^*$, $\boxed{x', f \mapsto listIter\, x' f\, \ell}$ determines a function $A' \to (A \to A' \to A') \to A'$ which is "polymorphic" in A' & A

Polymorphic lists

$$\alpha\, list \stackrel{\text{def}}{=} \forall\, \alpha' \, (\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha')$$

$$Nil \stackrel{\text{def}}{=} \Lambda\, \alpha, \alpha' \, (\lambda\, x' : \alpha', f : \alpha \to \alpha' \to \alpha' \, (x'))$$

$$\begin{aligned} Cons \stackrel{\text{def}}{=} \; & \Lambda\alpha(\lambda x : \alpha, \ell : \alpha\, list(\Lambda\alpha'(\\ & \quad \lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha'(\\ & \qquad f\, x\, (\ell\, \alpha'\, x'\, f))))) \end{aligned}$$

Polymorphic lists

$: \forall \alpha (\alpha\ list)$

$$\alpha\ list \stackrel{\text{def}}{=} \forall\, \alpha' \,(\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha')$$

$$Nil \stackrel{\text{def}}{=} \Lambda\, \alpha, \alpha' \,(\lambda\, x' : \alpha', f : \alpha \to \alpha' \to \alpha' \,(x'))$$

$$\begin{aligned} Cons \stackrel{\text{def}}{=} {} & \Lambda\alpha(\lambda x : \alpha, \ell : \alpha\ list(\Lambda\alpha'(\\ & \lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha'(\\ & f\, x\, (\ell\, \alpha'\, x'\, f))))) \end{aligned}$$

$: \forall \alpha (\alpha \to \alpha\, list \to \alpha\, list)$

List iteration in PLC

$$iter \stackrel{\text{def}}{=} \Lambda \alpha, \alpha' (\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha' ($$
$$\lambda \ell : \alpha\, list\, (\ell\, \alpha'\, x'\, f)))$$

satisfies:

- $\vdash iter : \forall\, \alpha, \alpha'\, (\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha\, list \to \alpha')$
- $iter\, \alpha\, \alpha'\, x'\, f\, (Nil\, \alpha) =_{\beta} x'$
- $iter\, \alpha\, \alpha'\, x'\, f\, (Cons\, \alpha\, x\, \ell) =_{\beta} f\, x\, (iter\, \alpha\, \alpha'\, x'\, f\, \ell)$

List iteration in PLC

$$iter \stackrel{\text{def}}{=} \Lambda\alpha, \alpha'(\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha'($$
$$\lambda \ell : \alpha\, list\, (\ell\, \alpha'\, x'\, f)))$$

satisfies:

- $\vdash iter : \forall\, \alpha, \alpha'\, (\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha\, list \to \alpha')$
- $iter\, \alpha\, \alpha'\, x'\, f\, (Nil\, \alpha) =_\beta x'$ $*$
- $iter\, \alpha\, \alpha'\, x'\, f\, (Cons\, \alpha\, x\, \ell) =_\beta f\, x\, (iter\, \alpha\, \alpha'\, x'\, f\, \ell)$

$\to^*$ $Nil\, \alpha\, \alpha'\, x'\, f$

List iteration in PLC

$$iter \stackrel{\text{def}}{=} \Lambda\alpha, \alpha'(\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha'($$
$$\lambda \ell : \alpha\, list\, (\ell\, \alpha'\, x'\, f)))$$

satisfies:

- $\vdash iter : \forall\, \alpha, \alpha'\, (\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha\, list \to \alpha')$
- $iter\, \alpha\, \alpha'\, x'\, f\, (Nil\, \alpha) =_\beta x'$
- $iter\, \alpha\, \alpha'\, x'\, f\, (Cons\, \alpha\, x\, \ell) =_\beta f\, x\, (iter\, \alpha\, \alpha'\, x'\, f\, \ell)$

$\to^* (Cons\, \alpha\, x\, \ell)\, \alpha'\, x'\, f$

$\to^* f\, x\, (\ell\, \alpha'\, x'\, f)$

List iteration in PLC

$$iter \stackrel{\text{def}}{=} \Lambda\alpha, \alpha'(\lambda x' : \alpha', f : \alpha \to \alpha' \to \alpha'(\lambda \ell : \alpha\, list\, (\ell\, \alpha'\, x'\, f)))$$

satisfies:

- $\vdash iter : \forall\, \alpha, \alpha'\, (\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha\, list \to \alpha')$
- $iter\, \alpha\, \alpha'\, x'\, f\, (Nil\, \alpha) =_\beta x'$
- $iter\, \alpha\, \alpha'\, x'\, f\, (Cons\, \alpha\, x\, \ell) =_\beta f\, x\, (iter\, \alpha\, \alpha'\, x'\, f\, \ell)$

$\ast\ (Cons\, \alpha\, x\, \ell)\, \alpha'\, x'\, f$

$\ast\ f\, x\, (\ell\, \alpha'\, x'\, f) \ \ast$

<u>FACT</u> Given a closed PLC type τ

$$\boxed{\begin{array}{l}\{\text{closed } \beta\text{-normal forms of type } \tau\,\mathrm{list}\} \\ \cong \\ \{\text{closed } \beta\text{-normal forms of type } \tau\}^*\end{array}}$$

$$\mathrm{nil} \longleftrightarrow \beta NF(Nil\,\tau)$$

$$N_1 :: \mathrm{nil} \longleftrightarrow \beta NF(Cons\,\tau\,(N_1\,(Nil\,\tau)))$$

$$N_2 :: N_1 :: \mathrm{nil} \longleftrightarrow \beta NF(Cons\,\tau\,(N_2\,(Cons\,\tau\,(N_1\,(Nil\,\tau)))))$$

etc

[Fig.5, p 59]

PLC encodings of ML algebraic datatypes

ML	PLC
$\alpha_1 * \alpha_2$	$\forall\alpha(((\alpha_1 \to \alpha_2 \to \alpha) \to \alpha)$
datatype (α_1, α_2)sum = Inl of α_1 | Inr of α_2	$\forall\alpha((\alpha_1 \to \alpha) \to (\alpha_2 \to \alpha) \to \alpha)$
datatype nat = Zero | Succ of nat	$\forall\alpha(\alpha \to (\alpha \to \alpha) \to \alpha)$
datatype binTree = Leaf | Node of binTree * binTree	$\forall\alpha(\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha)$

E.g. of a <u>non</u>-algebraic ML datatype

```
datatype nTree = Leaf
               | Node of (nat -> nTree)
```

[Section 5.2 , p 67]

Dependent Types

PLC syntax

Types	τ	$::=$	α	type variable
		$\mid$	$\tau \to \tau$	function type
		$\mid$	$\forall\, \alpha\, (\tau)$	$\forall$-type

Expressions

M	$::=$	x	variable
	$\mid$	$\lambda\, x : \tau\, (M)$	function abstraction
	$\mid$	$M\, M$	function application
	$\mid$	$\Lambda\, \alpha\, (M)$	type generalisation
	$\mid$	$M\, \tau$	type specialisation

(α and x range over fixed, countably infinite sets $\mathbf{TyVar}$ and $\mathbf{Var}$ respectively.)

expressions can "depend" on
(have free occurrences of) type variables

A tautology checker

$$\begin{aligned}\texttt{fun}\ taut\ x\ f = \texttt{if}\ x = 0\ \texttt{then}\ f\ \texttt{else}\ & (taut(x-1)(f\ \texttt{true}))\\ & \texttt{andalso}\ (taut(x-1)(f\ \texttt{false}))\end{aligned}$$

Defining types for each natural number $n \in \mathbb{N}$.

$$\begin{cases} 0\ AryBoolOp & \stackrel{\text{def}}{=}\ bool \\ (n+1)\ AryBoolOp & \stackrel{\text{def}}{=}\ bool \to (n\ AryBoolOp) \end{cases}$$

then $taut\ n$ has type $(n\ AryBoolOp) \to bool$, i.e. the result type of the function $taut$ *depends upon the value of its argument*.

E.g. $3\ AryBoolOp = \underbrace{bool \to bool \to bool}_{3\ \text{arguments}} \to bool$

The tautology checker in Agda

```
data Bool : Set where
  True : Bool
  False : Bool

_and_ : Bool -> Bool -> Bool
True and True = True
True and False = False
False and _ = False

data Nat : Set where
  Zero : Nat
  Succ : Nat -> Nat
```

```
_AryBoolOp : Nat -> Set
Zero AryBoolOp = Bool
(Succ n) AryBoolOp = Bool -> n AryBoolOp

taut : (n : Nat) -> n AryBoolOp -> Bool
taut Zero f = f
taut (Succ n) f = taut n (f True) and taut n (f False)
```

The tautology checker in Agda

```
data Bool : Set where
  True : Bool
  False : Bool

_and_ : Bool -> Bool -> Bool
True and True = True
True and False = False
False and _ = False

data Nat : Set where
  Zero : Nat
  Succ : Nat -> Nat
```

```
_AryBoolOp : Nat -> Set
Zero AryBoolOp = Bool
(Succ n) AryBoolOp = Bool -> n AryBoolOp

taut : (n : Nat) -> n AryBoolOp -> Bool
taut Zero f = f
taut (Succ n) f = taut n (f True) and taut n (f False)
```

e.g. of a dependent function type

Dependent function types $(x : \tau) \to \tau'$

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda\, x : \tau\,(M) : (x : \tau) \to \tau'} \quad \text{if } x \notin dom(\Gamma) \cup fv(\Gamma)$$

$$\frac{\Gamma \vdash M : (x : \tau) \to \tau' \quad \Gamma \vdash M' : \tau}{\Gamma \vdash M\, M' : \tau'[M'/x]}$$

τ' may ‘depend’ on x, i.e. have free occurrences of x.

(Free occurrences of x in τ' are bound in $(x : \tau) \to \tau'$.)

Dependent type systems feature rules like

$$\frac{\Gamma \vdash M : \tau \qquad \tau \approx \tau'}{\Gamma \vdash M : \tau'}$$

(E.g. $(1+1)\mathrm{AnyBoolOp} \approx 2\,\mathrm{AnyBoolOp}$)

For decidability, need $\tau \approx \tau'$ to be a decidable relation between type expressions.