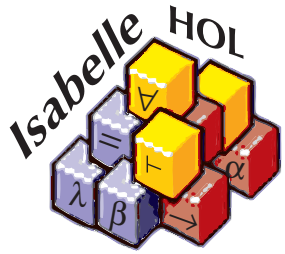
Inductive datatypes in HOL

—

lessons learned in Formal-Logic Engineering

Stefan Berghofer and Markus Wenzel

Institut für Informatik
TU München



Introduction

Applications of inductive datatypes

- Data structures (lists, trees, ...)
- Abstract syntax of programming languages (e.g. Bali)
- ...

What we want

simple user-supplied datatype specification $\longrightarrow$ [datatype package] $\longrightarrow$ characteristic properties (induction, recursion, ...)

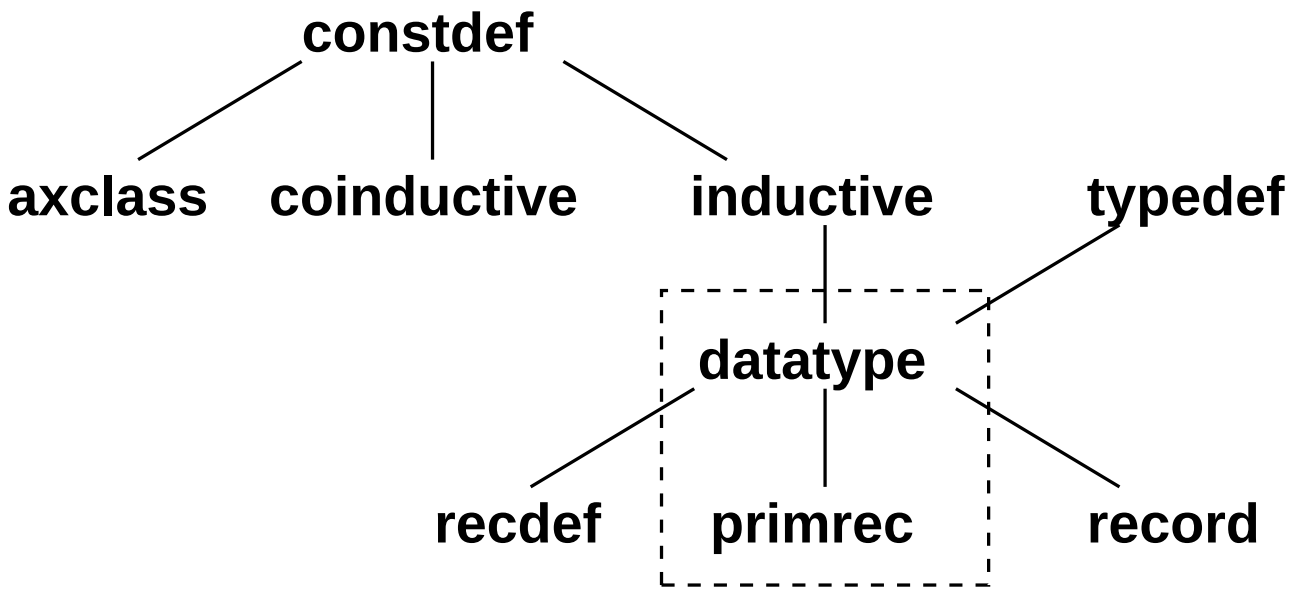
Different approaches

Axiomatic Desired theorems are introduced as axioms
($\Rightarrow$ danger of introducing inconsistencies)

Inherent Concept of inductive definitions is part of the logic (e.g. Coq)

Definitional Define datatypes in terms of already existing types, derive characteristic theorems from definition by formal proof

Definitional packages in Isabelle/HOL



Examples

datatype α aexp = If (α bexp) (α aexp) (α aexp)
| Sum (α aexp) (α aexp)
| Var α
| Num nat
and α bexp = Less (α aexp) (α aexp)
| And (α bexp) (α bexp)

datatype (α, β)term = Var α
| App β (((α, β)term)list)

datatype (α, β, γ)tree = Atom α
| Branch β ($\gamma \to (\alpha, \beta, \gamma)$tree)

datatype α lambda = Var α
| App (α lambda) (α lambda)
| Lam ($\alpha \to \alpha$ lambda)

Limitations of set-theoretic datatypes

Cardinality restrictions (Cantor)

datatype t = C (t → bool) | D ((bool → t) → bool) | E ((t → bool) → bool) | F

C, D, E would yield injections of type (t → bool) ↪ t

"Dangerous" type parameters

datatype (α, β, γ)t = C (α set) | D (β → γ) | E (γ list)

Caution: α and β are dangerous:

- α occurs as an argument of non-datatype constructor "set"
- β occurs as left argument of "→" (i.e. not *strictly positive*)

datatype δ u = F ((δ u, δ, δ)t) | G

datatype δ u = F ((δ, δ u, δ)t) | G

datatype δ u = F ((δ, δ, δ u)t) | G

Admissible datatype specifications

$$\begin{array}{lllll}
\textbf{datatype} & (\alpha_1,\ldots,\alpha_h)t_1 & = & C_1^1\ \tau_{1,1}^1\ \ldots\ \tau_{1,m_1^1}^1 \mid \ldots \mid C_{k_1}^1\ \tau_{k_1,1}^1\ \ldots\ \tau_{k_1,m_{k_1}^1}^1 \\
 & & \ldots & \\
\textbf{and} & (\alpha_1,\ldots,\alpha_h)t_n & = & C_1^n\ \tau_{1,1}^n\ \ldots\ \tau_{1,m_1^n}^n \mid \ldots \mid C_{k_n}^n\ \tau_{k_n,1}^n\ \ldots\ \tau_{k_n,m_{k_n}^n}^n
\end{array}$$

Types $\tau_{i,i'}^j$ must be admissible. A type τ is admissible iff

- τ is non-recursive (i.e. does not contain $t_1,\ldots,t_n$), or
- $\tau = \sigma \to \tau'$, where σ is non-recursive and τ' is admissible, or
- $\tau = (\alpha_1,\ldots,\alpha_h)t_{j'}$, where $1 \le j' \le n$, or
- $\tau = (\tau'_1,\ldots,\tau'_{h'})t'$, where
  - t' is an existing datatype, and
  - $\tau'_1,\ldots,\tau'_{h'}$ are admissible, and
  - if τ'_i is recursive, then ith argument of t' is *not* dangerous

A universe for recursive types

Observation: elements of datatypes are trees

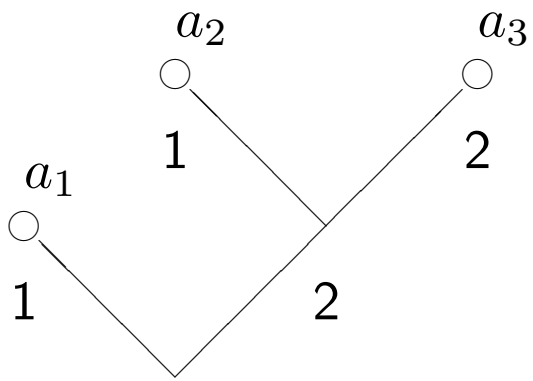
Leaf :: $\alpha \to (\alpha, \beta)\text{dtree}$	embedding non-recursive occurrences of types
In0, In1 :: $(\alpha, \beta)\text{dtree} \to (\alpha, \beta)\text{dtree}$	modeling distinct constructors
Pair :: $(\alpha, \beta)\text{dtree} \to (\alpha, \beta)\text{dtree} \to (\alpha, \beta)\text{dtree}$	modeling constructors with multiple arguments
Lim :: $(\beta \to (\alpha, \beta)\text{dtree}) \to (\alpha, \beta)\text{dtree}$	embedding function types (infinitary products)

Representing trees in HOL

$$(\alpha, \beta)\text{node} = \underbrace{(\overbrace{\text{nat}}^{\text{position}} \to \overbrace{(\beta + \text{nat})}^{\text{branching label}})}_{\text{path}} \times \underbrace{(\alpha + \text{bool})}_{\text{node value}}$$

$$(\alpha, \beta)\text{dtree} = ((\alpha, \beta)\text{node})\text{set}$$

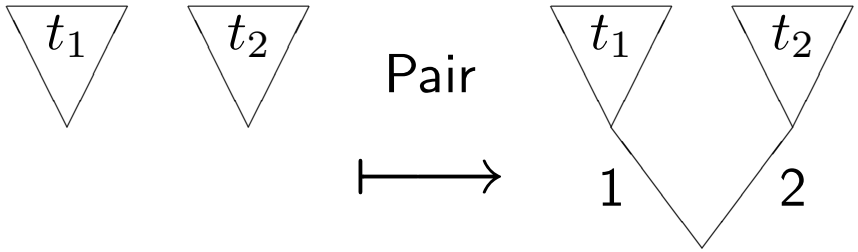
Representing trees in HOL



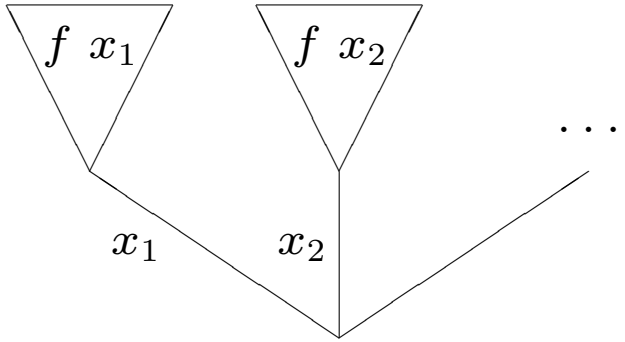
$$t = \{(f_1, a_1), (f_2, a_2), (f_3, a_3)\}$$

where

$$\begin{aligned} f_1 &= \langle 1, 0, 0, \ldots \rangle \\ f_2 &= \langle 2, 1, 0, 0, \ldots \rangle \\ f_3 &= \langle 2, 2, 0, 0, \ldots \rangle \end{aligned}$$



Pair t_1 t_2 $\equiv$ (push (Inr 1) “ t_1) $\cup$ (push (Inr 2) “ t_2)



Lim $f \equiv \bigcup\{z \mid \exists x.\ z = \text{push (Inl } x) \text{ “ } (f\ x)\}$

Constructing an example datatype — α list

datatype α list = Nil | Cons α (α list)

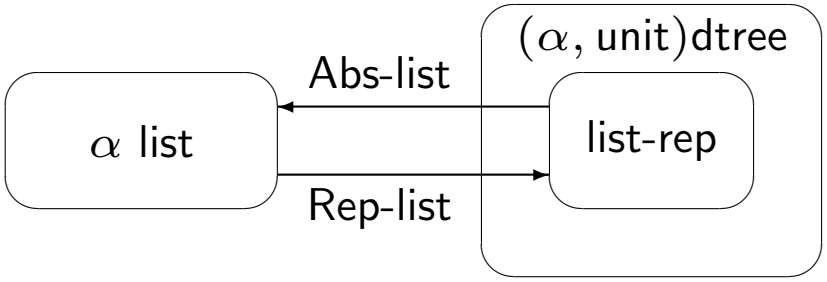
Inductive definition of the representing set

$$\frac{}{\text{Nil-rep} \in \text{list-rep}} \qquad \frac{ys \in \text{list-rep}}{\text{Cons-rep } y\ ys \in \text{list-rep}}$$

Nil-rep	$\equiv$	In0 dummy
Cons-rep y ys	$\equiv$	In1 (Pair (Leaf y) ys)

HOL type definition

$\vdash$ Abs-list (Rep-list x) $= x$
$\vdash y \in$ list-rep $\Longrightarrow$ Rep-list (Abs-list y) $= y$
$\vdash$ Rep-list $x \in$ list-rep



Constructors

Nil	$\equiv$	Abs-list Nil-rep
Cons x xs	$\equiv$	Abs-list (Cons-rep x (Rep-list xs))

Nil $\neq$ Cons x xs because In0 $t \neq$ In1 t'

Cons x xs = Cons y ys $\Longleftrightarrow$ $x = y \wedge xs = ys$ because In1, Pair, Leaf, Abs-list and Rep-list are injective

Inductive definitions and least fixpoints

$$\text{lfp } F = \bigcap\{x \mid F\ x \subseteq x\}$$

The Knaster-Tarski theorem

If F is monotone then

$$\text{lfp } F = F\ (\text{lfp } F)$$

$$\frac{F\ y \subseteq y}{\text{lfp } F \subseteq y}$$

$$\frac{F\ \{x \mid P\ x\} \subseteq \{x \mid P\ x\}}{\text{lfp } F \subseteq \{x \mid P\ x\}}\ \text{(weakInd)} \qquad \frac{F\ (\text{lfp } F \cap \{x \mid P\ x\}) \subseteq \{x \mid P\ x\}}{\text{lfp } F \subseteq \{x \mid P\ x\}}\ \text{(strongInd)}$$

list-rep as a least fixpoint

$$F\ L = \{x \mid x = \text{Nil-rep} \vee \exists y\ ys.\ x = \text{Cons-rep}\ y\ ys \wedge ys \in L\}$$

$$\text{list-rep} = \text{lfp } F$$

$$\textbf{datatype}\ \alpha\ \text{list} = \text{Nil} \mid \text{Cons}\ \alpha\ (\alpha\ \text{list}) \quad \rightsquigarrow \quad \text{list-rep} = F\ \text{list-rep}$$

Structural induction — α list

Induction rule for type α list

$$\frac{P\ \mathsf{Nil} \quad \forall x\ xs.\ P\ xs \Longrightarrow P\ (\mathsf{Cons}\ x\ xs)}{P\ xs}\ (\mathit{list\text{-}ind})$$

Induction rule for list-rep

$$\frac{Q\ \mathsf{Nil\text{-}rep} \quad \forall y\ ys.\ Q\ ys \wedge ys \in \mathsf{list\text{-}rep} \Longrightarrow Q\ (\mathsf{Cons\text{-}rep}\ y\ ys)}{ys \in \mathsf{list\text{-}rep} \Longrightarrow Q\ ys}\ (\mathit{list\text{-}rep\text{-}ind})$$

Derivation of *list-ind* from *list-rep-ind*

$$\dfrac{\dfrac{\dots \Longrightarrow P\ (\mathsf{Cons}\ y\ (\mathsf{Abs\text{-}list}\ ys))}{\dots \Longrightarrow P\ (\mathsf{Abs\text{-}list}\ (\mathsf{Cons\text{-}rep}\ y\ (\mathsf{Rep\text{-}list}\ (\mathsf{Abs\text{-}list}\ ys))))} \quad \dots \quad \forall y\ ys.\ P\ (\mathsf{Abs\text{-}list}\ ys) \wedge ys \in \mathsf{list\text{-}rep} \Longrightarrow P\ (\mathsf{Abs\text{-}list}\ (\mathsf{Cons\text{-}rep}\ y\ ys))}{\mathsf{Rep\text{-}list}\ xs \in \mathsf{list\text{-}rep} \Longrightarrow P\ (\mathsf{Abs\text{-}list}\ (\mathsf{Rep\text{-}list}\ xs))}}{P\ xs}$$

Primitive recursion — α list

Recursion combinator

$$\text{list-rec} :: \beta \to (\alpha \to \alpha\ \text{list} \to \beta \to \beta) \to \alpha\ \text{list} \to \beta$$
$$\text{list-rec}\ f_1\ f_2\ \text{Nil} = f_1$$
$$\text{list-rec}\ f_1\ f_2\ (\text{Cons}\ x\ xs) = f_2\ x\ xs\ (\text{list-rec}\ f_1\ f_2\ xs)$$

Example

$$\begin{array}{lcl} \text{foldl}\ f\ a\ \text{Nil} & = & a \\ \text{foldl}\ f\ a\ (\text{Cons}\ x\ xs) & = & \text{foldl}\ f\ (f\ a\ x)\ xs \end{array}$$

can be defined by

$$\text{foldl} \equiv \lambda f'\ a'\ xs'.\ \text{list-rec}\ (\lambda f\ a.\ a)\ (\lambda x\ xs\ r.\ (\lambda f\ a.\ r\ f\ (f\ a\ x)))\ xs'\ f'\ a'$$

Inductive definition of function graph list-rel

$$\frac{}{(\text{Nil},\ f_1) \in \text{list-rel}\ f_1\ f_2} \qquad \frac{(xs,\ y) \in \text{list-rel}\ f_1\ f_2}{(\text{Cons}\ x\ xs,\ f_2\ x\ xs\ y) \in \text{list-rel}\ f_1\ f_2}$$

$$\text{list-rec}\ f_1\ f_2\ xs \equiv \varepsilon y.\ (xs,\ y) \in \text{list-rel}\ f_1\ f_2$$

show: $\exists! y.\ (xs,\ y) \in \text{list-rel}\ f_1\ f_2$ (by induction on xs)

Datatypes with nested recursion — (α, β)term

"Unfolding" the datatype specification

$$\begin{array}{lllll} \textbf{datatype} & (\alpha,\beta)\text{term} & = & \text{Var } \alpha \mid \text{App } \beta\ ((\alpha,\beta)\text{term-list}) \\ \textbf{and} & (\alpha,\beta)\text{term-list} & = & \text{Nil}' \mid \text{Cons}'\ ((\alpha,\beta)\text{term})\ ((\alpha,\beta)\text{term-list}) \end{array}$$

The representing sets

term-rep, term-list-rep :: $((\alpha + \beta, \text{unit})\text{dtree})\text{set})$

$$\frac{}{\text{In0 (Leaf (Inl } a)) \in \text{term-rep}} \qquad \frac{ts \in \text{term-list-rep}}{\text{In1 (Pair (Leaf (Inr } b))\ ts) \in \text{term-rep}}$$

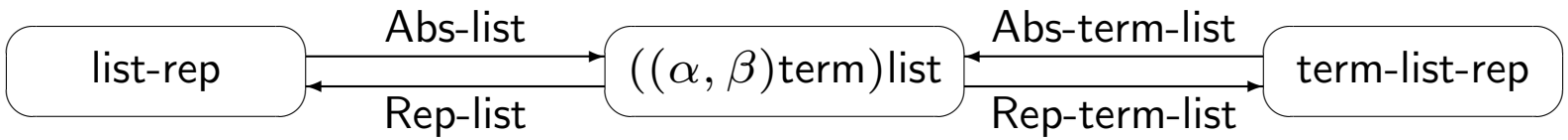
$$\frac{}{\text{In0 dummy} \in \text{term-list-rep}} \qquad \frac{t \in \text{term-rep} \quad ts \in \text{term-list-rep}}{\text{In1 (Pair } t\ ts) \in \text{term-list-rep}}$$

Representation function (primitive recursive)

Rep-term-list Nil $=$ In0 dummy

Rep-term-list (Cons $t\ ts$) $=$ In1 (Pair (Rep-term t) (Rep-term-list ts))

Abs-term-list $\equiv$ inv Rep-term-list



(α, β)term **continued**

Constructors

$$\begin{array}{lcl} \text{Var } a & \equiv & \text{Abs-term (In0 (Leaf (Inl } a\text{)))} \\ \text{App } b\ ts & \equiv & \text{Abs-term (In1 (Pair (Leaf (Inr } b\text{)) (Rep-term-list } ts\text{)))} \end{array}$$

Induction

$$\frac{\forall a.\ P\ (\text{Var } a) \quad \forall b\ ts.\ Q\ ts \Longrightarrow P\ (\text{App } b\ ts) \quad Q\ \text{Nil} \quad \forall t\ ts.\ P\ t \wedge Q\ ts \Longrightarrow Q\ (\text{Cons } t\ ts)}{P\ t \wedge Q\ ts}$$

Primitive recursion

$$\begin{array}{lcl} \text{term-rec } f_1 \ldots f_4\ (\text{Var } a) & = & f_1\ a \\ \text{term-rec } f_1 \ldots f_4\ (\text{App } b\ ts) & = & f_2\ b\ ts\ (\text{term-list-rec } f_1 \ldots f_4\ ts) \\ \text{term-list-rec } f_1 \ldots f_4\ \text{Nil} & = & f_3 \\ \text{term-list-rec } f_1 \ldots f_4\ (\text{Cons } t\ ts) & = & f_4\ t\ ts\ (\text{term-rec } f_1 \ldots f_4\ t)\ (\text{term-list-rec } f_1 \ldots f_4\ ts) \end{array}$$

Infinitely branching datatypes — (α, β, γ)tree

The representing set tree-rep :: $((\alpha + \beta, \gamma)\text{dtree})\text{set}$

$$\frac{}{\text{In0 (Leaf (Inl } a)) \in \text{tree-rep}} \qquad \frac{g \in \text{Funs tree-rep}}{\text{In1 (Pair (Leaf (Inr } b)) \text{ (Lim } g)) \in \text{tree-rep}}$$

where

Funs :: β set $\longrightarrow (\alpha \longrightarrow \beta)$set

Funs $S \equiv \{g \mid \text{range } g \subseteq S\}$

Constructors

Atom $a \equiv$ Abs-tree (In0 (Leaf (Inl a)))

Branch $b\ f \equiv$ Abs-tree (In1 (Pair (Leaf (Inr b)) (Lim (Rep-tree $\circ$ f))))

Structural induction

$$\frac{\forall a.\ P\ (\text{Atom } a) \quad \forall b\ f.\ (\forall x.\ P\ (f\ x)) \Longrightarrow P\ (\text{Branch } b\ f)}{P\ t}$$

Primitive recursion — (α, β, γ)tree

Recursion combinator

$\text{tree-rec} :: (\alpha \to \delta) \to (\beta \to (\gamma \to (\alpha, \beta, \gamma)\text{tree}) \to (\gamma \to \delta) \to \delta) \to (\alpha, \beta, \gamma)\text{tree} \to \delta$

$\text{tree-rec}\ f_1\ f_2\ (\text{Atom}\ a) = f_1\ a$

$\text{tree-rec}\ f_1\ f_2\ (\text{Branch}\ b\ f) = f_2\ b\ f\ ((\text{tree-rec}\ f_1\ f_2) \circ f)$

Example

$$\begin{array}{lcl} \text{member}\ c\ (\text{Atom}\ a) & = & (a = c) \\ \text{member}\ c\ (\text{Branch}\ b\ f) & = & (\exists x.\ \text{member}\ c\ (f\ x)) \end{array}$$

can be defined by

$\text{member} \equiv \lambda c.\ \text{tree-rec}\ (\lambda a.\ a = c)\ (\lambda b\ f\ f'.\ \exists x.\ f'\ x)$

Inductive definition of function graph tree-rel

$$\frac{}{(\text{Atom}\ a,\ f_1\ a) \in \text{tree-rel}\ f_1\ f_2} \qquad \frac{f' \in \text{compose}\ f\ (\text{tree-rel}\ f_1\ f_2)}{(\text{Branch}\ b\ f,\ f_2\ b\ f\ f') \in \text{tree-rel}\ f_1\ f_2}$$

where

$$\begin{array}{lcl} \text{compose} & :: & (\alpha \to \beta) \to (\beta \times \gamma)\text{set} \to (\alpha \to \gamma)\text{set} \\ \text{compose}\ f\ R & \equiv & \{f' \mid \forall x.\ (f\ x,\ f'\ x) \in R\} \end{array}$$

Lessons learned

- Approaches to theory extension mechanisms
  $\Rightarrow$ definitional
- The logic of choice?
  $\Rightarrow$ take HOL *as is* ($\approx$ simply typed, classical set theory)
- Layered arrangements of concepts ($\approx$ deep vs. shallow embedding)
  - extra-logical notion of admissibility
  - extra-logical "unfolding"
- System integration
  - Bootstrapping problem: universe dtree depends on types nat, $\alpha + \beta$, $\alpha \times \beta$
    $\Rightarrow$ Represent these types as datatypes later!
    rep-datatype: $\left\{ \begin{array}{l} \text{freeness,} \\ \text{induction} \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} \text{primitive recursion,} \\ \text{representation function} \end{array} \right\}$
    $\Rightarrow$ More uniform treatment of types
  - Cooperation of packages: not just logical theory extension, also additional information ("theory data" concept)

Conclusion

Achievements

- Set theoretic construction of
  - mutually recursive
  - nested
  - arbitrarily branching

  types, together with primitive recursion
- Knaster-Tarski fixpoint approach
- Scalable and robust working environment

Further work

- Actual combination of definitional packages
- Codatatypes (mixing with datatypes?)
- Non-freely generated types:

  implement $\quad \mathbf{datatype}\ \alpha\ \mathsf{t} = \mathsf{C}\ \alpha \mid \mathsf{D}\ ((\alpha\ \mathsf{t})\mathsf{finset})$

  by $\quad \mathbf{datatype}\ \alpha\ \mathsf{t} = \mathsf{C}\ \alpha \mid \mathsf{D}\ ((\alpha\ \mathsf{t})\mathsf{list})$

  $\Rightarrow$ quotient construction