Polymorphic Variants in Dependent Type Theory

Dominic Mulligan <mulligan@cs.unibo.it>
Claudio Sacerdoti Coen <sacerdot@cs.unibo.it>

University of Cambridge — University of Bologna

13/05/2014

The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

Outline

Outline

The Expression Problem

Additive expressions:

inductive Expr :=
  Const: $\mathbb{N} \to$ Expr
| Plus: Expr $\to$ Expr $\to$ Expr

let rec eval (E:Expr) : $\mathbb{N}$:=
match E **with**
  Const n $\Rightarrow$ n
| Plus E1 E2 $\Rightarrow$ eval E1 + eval E2

Multiplicative expressions:

inductive Expr :=
  Const: $\mathbb{N} \to$ Expr
| Mult: Expr $\to$ Expr $\to$ Expr

let rec eval (E:Expr) : $\mathbb{N}$:=
match E **with**
  Const n $\Rightarrow$ n
| Mult E1 E2 $\Rightarrow$ eval E1 $*$ eval E2

How to
merge the two datatypes and functions
reusing the code?

Merge = recursive coalesced sum

The Dual Problem: Extensible Records

Pretty-printing status :

record status :=
{ expr: *Expr*
; width: $\mathbb{N}$
}

let set_width (st:status) w :=
st.width $<-$ w

Evaluation status:

record status :=
{ expr: *Expr*
; callback: $\mathbb{N} \to$ *unit*
}

let chain_cb (st:status) f :=
st.callback $<-$ st.callback $\circ$ f

How to
merge the two records and functions
reusing the code?

Merge = coalesced product

Solving the Expression Problem

1. Temporarily break the recursion
2. Implement the coalesced sums/products

Temporarily breaking the recursion

- Lazy binding (OO languages)

```
class Plus implements Expr =
{ fst : Expr
; snd: Expr
; method eval = fst.eval + snd.eval
}
```

- Monotone functors + Recursive Types (FL)

```
inductive Expr (T) :=
    Const: ℕ → Expr
  | Plus: T → T → Expr

type AddExpr := Expr AddExpr
```

```
let eval (f: T → ℕ) (E:Expr T) : ℕ :=
 match E with
    Const n ⇒ n
  | Plus E1 E2 ⇒ f E1 + f E2

let rec eval_add := eval eval_add
```

Implement the coalesced sums/products

- First solution: coalesced sum = disjoint sum + quotienting

```
type t1 = A of ℕ | B of 𝔹
type t2 = A of ℕ | C

type t1_plus_t2 = Inl of t1 | Inr of t2

let print_t1_plus_t2 :=
 function
     Inl x ⇒ print_t1 x
   | Inr y ⇒ print_t2 y
```

Cons:

1. Inl (A 2)$\simeq$Inr (A 2) need quotienting (???) or refactoring
   what if print_t1 (A 2) $\neq$ print_t2 (A 2)?
2. non associative: $(t1 + t2) + t3 \neq t1 + (t2 + t3)$
3. Inefficient memory representation and pattern matching

Implement the coalesced sums/products

- First solution: coalesced sum = disjoint sum + quotienting

```
type t1 = A of ℕ | B of 𝔹
type t2 = A of ℕ | C

type t1_plus_t2 = Inl of t1 | Inr of t2

let print_t1_plus_t2 :=
 function
     Inl x ⇒ print_t1 x
   | Inr y ⇒ print_t2 y
```

Pros:

1. the sum can always be performed
2. separate compilation (but refactoring defeats it)
3. no change to the language

Implement the coalesced sums/products

- Second solution: Garrigue's Polymorphic Variants (dually, Remy's Extensible Records)

```
type t1 = [‘A of ℕ | ‘B of 𝔹]   type t2 = [‘A of ℕ | ‘C]

type t1_plus_t2 = [t1 | t2]

let print_t1_plus_t2 : [< ‘A of ℕ | ‘B of 𝔹 | ‘C] → ℕ :=
 function
     (‘A _ | ‘B _) as x ⇒ print_t1 x
   | (‘C) as y ⇒ print_t2 y
```

Pros:

1. separate compilation
2. order of patterns decides the behaviour
3. efficient memory representation and pattern matching (on modern hardware)

Implement the coalesced sums/products

- Second solution: Garrigue's Polymorphic Variants (dually, Remy's Extensible Records)

```
type t1 = [‘A of ℕ | ‘B of 𝔹]   type t2 = [‘A of ℕ | ‘C]

type t1_plus_t2 = [t1 | t2]

let print_t1_plus_t2 : [< ‘A of ℕ | ‘B of 𝔹 | ‘C] → ℕ :=
 function
     (‘A _ | ‘B _) as x ⇒ print_t1 x
   | (‘C) as y ⇒ print_t2 y
```

Cons:

1. sums may fail at compile time (very unlikely)
2. complex type system and type inference (e.g. [> ‘A of ℕ < ‘A of ℕ | ‘B of 𝔹])

Goal of the talk

Can we implement in type theory (Matita/Coq) polymorphic variants/extensible records so that:

1. no changes to the language
2. separate compilation
3. no refactoring
4. the extracted code is as memory efficient as standard inductive types
5. subtyping coercions are extracted to identity

We solve the problem for bounded variants/records giving away separate compilation and a bit of efficiency for polymorphic records.

Outline

Polymorphic variants

Syntax of types:

$$
\begin{array}{rcl}
T & ::= & \ldots \mid [i|S] \mid \forall i_{\geq\langle L,U\rangle}.T \mid \forall \rho.T \\
S & ::= & \emptyset \mid tag : T, S \mid \rho \\
L, U & ::= & \emptyset \mid i, L \\
K & ::= & \emptyset \mid K \oplus i_{\geq\langle L,U\rangle}
\end{array}
$$

1. i specifies what tags (constructors) inhabit the type
2. S maps tags to their type

Polymorphic variants

Syntax of types:

$$
\begin{array}{rcl}
T & ::= & \ldots \mid [i|S] \mid \forall i_{\geq \langle L,U \rangle}.T \mid \forall \rho.T \\
S & ::= & \emptyset \mid tag : T, S \mid \rho \\
L, U & ::= & \emptyset \mid i, L \\
K & ::= & \emptyset \mid K \oplus i_{\geq \langle L,U \rangle}
\end{array}
$$

- S is a list or declarations

Polymorphic variants

Syntax of types:

$$
\begin{array}{rcl}
T & ::= & \ldots \mid [i|S] \mid \forall i_{\geq \langle L,U \rangle}.T \mid \forall \rho.T \\
S & ::= & \emptyset \mid tag : T, S \mid \rho \\
L, U & ::= & \emptyset \mid i, L \\
K & ::= & \emptyset \mid K \oplus i_{\geq \langle L,U \rangle}
\end{array}
$$

- S is a list or declarations
- the type is parametrically polymorphic on the types of additional tags (specified by ρ)

Polymorphic variants

Syntax of types:

$$
\begin{array}{rcl}
T & ::= & \ldots \mid [i|S] \mid \forall i_{\geq \langle L,U \rangle}.T \mid \forall \rho.T \\
S & ::= & \emptyset \mid tag : T, S \mid \rho \\
L, U & ::= & \emptyset \mid i, L \\
K & ::= & \emptyset \mid K \oplus i_{\geq \langle L,U \rangle}
\end{array}
$$

- i specifies what tags (constructors) inhabit the type via bounded parametric polymorphism
- $i \geq \langle L, U \rangle$ ranges over sets A of tags such that $L \subseteq A \subseteq U$

Polymorphic variants

Syntax of types:

$$
\begin{array}{rcl}
T & ::= & \ldots \mid [i|S] \mid \forall i_{\geq \langle L,U \rangle}.T \mid \forall \rho.T \\
S & ::= & \emptyset \mid tag : T, S \mid \rho \\
L, U & ::= & \emptyset \mid i, L \\
K & ::= & \emptyset \mid K \oplus i_{\geq \langle L,U \rangle}
\end{array}
$$

- K is a context of constraints

Polymorphic variants

$$\frac{(L \subseteq L' \quad H \subseteq H')}{K \oplus i_{\geq \langle L,U \rangle} \vdash i \geq \langle L', H' \rangle}\ (\textbf{Ki}) \qquad \frac{K; \Gamma \vdash r : \phi \quad K \vdash i \geq \langle tag, \top \rangle}{K; \Gamma \vdash \text{`}tag(r) : [i \mid tag : \phi, T]}\ (\textbf{Kvariant})$$

$$\frac{K; \Gamma \vdash r : [i \mid \{tag_k : \phi_k\}_1^n, T] \quad K \vdash i \geq \langle \bot, \{tag_k\}_1^n \rangle \quad K; \Gamma, x_k : \phi_k \vdash r_k : \psi \quad (1 \leq k \leq n)}{K; \Gamma \vdash \textbf{match } r \textbf{ with } \{\text{`}tag_k(x_k) \Rightarrow r_k\}_1^n : \psi}\ (\textbf{Kmatch})$$

$$\frac{K; \Gamma \vdash r : \sigma \quad (\rho \notin ftv(\Gamma))}{K; \Gamma \vdash r : \forall \rho . \sigma}\ (\textbf{K}\forall\rho) \qquad \frac{K; \Gamma \vdash r : \forall \rho . \sigma \quad K; \Gamma \vdash s : T}{K; \Gamma \vdash rs : \sigma[\rho := T]}\ (\textbf{Kinst}\rho)$$

$$\frac{K \oplus i_{\geq \langle L,U \rangle}; \Gamma \vdash r : \sigma}{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma}\ (\textbf{K}\forall\textbf{i}) \qquad \frac{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma \quad K \vdash i' \geq \langle L, U \rangle}{K; \Gamma \vdash r : \sigma[i := i']}\ (\textbf{Kinsti})$$

Polymorphic variants

$$\frac{(L \subseteq L' \quad H \subseteq H')}{K \oplus i_{\geq \langle L,U \rangle} \vdash i \geq \langle L', H' \rangle}\ (\textbf{Ki}) \qquad \frac{K; \Gamma \vdash r : \phi \quad K \vdash i \geq \langle tag, \top \rangle}{K; \Gamma \vdash \text{`}tag(r) : [i \mid tag : \phi, T]}\ (\textbf{Kvariant})$$

$$\frac{K; \Gamma \vdash r : [i \mid \{tag_k : \phi_k\}_1^n, T] \quad K \vdash i \geq \langle \bot, \{tag_k\}_1^n \rangle \quad K; \Gamma, x_k : \phi_k \vdash r_k : \psi \quad (1 \leq k \leq n)}{K; \Gamma \vdash \textbf{match } r \textbf{ with } \{\text{`}tag_k(x_k) \Rightarrow r_k\}_1^n : \psi}\ (\textbf{Kmatch})$$

$$\frac{K; \Gamma \vdash r : \sigma \quad (\rho \notin ftv(\Gamma))}{K; \Gamma \vdash r : \forall \rho . \sigma}\ (\textbf{K}\forall\rho) \qquad \frac{K; \Gamma \vdash r : \forall \rho . \sigma \quad K; \Gamma \vdash s : T}{K; \Gamma \vdash rs : \sigma[\rho := T]}\ (\textbf{Kinst}\rho)$$

$$\frac{K \oplus i_{\geq \langle L,U \rangle}; \Gamma \vdash r : \sigma}{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma}\ (\textbf{K}\forall\textbf{i}) \qquad \frac{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma \quad K \vdash i' \geq \langle L, U \rangle}{K; \Gamma \vdash r : \sigma[i := i']}\ (\textbf{Kinsti})$$

Polymorphic variants

$$\frac{(L \subseteq L' \quad H \subseteq H')}{K \oplus i_{\geq \langle L,U \rangle} \vdash i \geq \langle L', H' \rangle}\ (\textbf{Ki}) \qquad \frac{K; \Gamma \vdash r : \phi \quad K \vdash i \geq \langle tag, \top \rangle}{K; \Gamma \vdash \text{`}tag(r) : [i \mid tag : \phi, T]}\ (\textbf{Kvariant})$$

$$\frac{K; \Gamma \vdash r : [i \mid \{tag_k : \phi_k\}_1^n, T] \quad K \vdash i \geq \langle \bot, \{tag_k\}_1^n \rangle \quad K; \Gamma, x_k : \phi_k \vdash r_k : \psi \quad (1 \leq k \leq n)}{K; \Gamma \vdash \textbf{match } r \textbf{ with } \{\text{`}tag_k(x_k) \Rightarrow r_k\}_1^n : \psi}\ (\textbf{Kmatch})$$

$$\frac{K; \Gamma \vdash r : \sigma \quad (\rho \notin ftv(\Gamma))}{K; \Gamma \vdash r : \forall \rho . \sigma}\ (\textbf{K}\forall\rho) \qquad \frac{K; \Gamma \vdash r : \forall \rho . \sigma \quad K; \Gamma \vdash s : T}{K; \Gamma \vdash rs : \sigma[\rho := T]}\ (\textbf{Kinst}\rho)$$

$$\frac{K \oplus i_{\geq \langle L,U \rangle}; \Gamma \vdash r : \sigma}{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma}\ (\textbf{K}\forall\textbf{i}) \qquad \frac{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma \quad K \vdash i' \geq \langle L, U \rangle}{K; \Gamma \vdash r : \sigma[i := i']}\ (\textbf{Kinsti})$$

Polymorphic variants

$$\frac{(L \subseteq L' \quad H \subseteq H')}{K \oplus i_{\geq \langle L,U \rangle} \vdash i \geq \langle L', H' \rangle}\ (\textbf{Ki}) \qquad \frac{K; \Gamma \vdash r : \phi \quad K \vdash i \geq \langle tag, \top \rangle}{K; \Gamma \vdash \text{`}tag(r) : [i \mid tag : \phi, T]}\ (\textbf{Kvariant})$$

$$\frac{K; \Gamma \vdash r : [i \mid \{tag_k : \phi_k\}_1^n, T] \quad K \vdash i \geq \langle \bot, \{tag_k\}_1^n \rangle \quad K; \Gamma, x_k : \phi_k \vdash r_k : \psi \quad (1 \leq k \leq n)}{K; \Gamma \vdash \textbf{match } r \textbf{ with } \{\text{`}tag_k(x_k) \Rightarrow r_k\}_1^n : \psi}\ (\textbf{Kmatch})$$

$$\frac{K; \Gamma \vdash r : \sigma \quad (\rho \notin ftv(\Gamma))}{K; \Gamma \vdash r : \forall \rho . \sigma}\ (\textbf{K}\forall\rho) \qquad \frac{K; \Gamma \vdash r : \forall \rho . \sigma \quad K; \Gamma \vdash s : T}{K; \Gamma \vdash rs : \sigma[\rho := T]}\ (\textbf{Kinst}\rho)$$

$$\frac{K \oplus i_{\geq \langle L,U \rangle}; \Gamma \vdash r : \sigma}{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma}\ (\textbf{K}\forall\textbf{i}) \qquad \frac{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma \quad K \vdash i' \geq \langle L, U \rangle}{K; \Gamma \vdash r : \sigma[i := i']}\ (\textbf{Kinsti})$$

Polymorphic variants

$$\frac{(L \subseteq L' \quad H \subseteq H')}{K \oplus i_{\geq \langle L,U \rangle} \vdash i \geq \langle L', H' \rangle}\ (\textbf{Ki}) \qquad \frac{K; \Gamma \vdash r : \phi \quad K \vdash i \geq \langle tag, \top \rangle}{K; \Gamma \vdash \text{`}tag(r) : [i \mid tag : \phi, T]}\ (\textbf{Kvariant})$$

$$\frac{K; \Gamma \vdash r : [i \mid \{tag_k : \phi_k\}_1^n, T] \quad K \vdash i \geq \langle \bot, \{tag_k\}_1^n \rangle \quad K; \Gamma, x_k : \phi_k \vdash r_k : \psi \quad (1 \leq k \leq n)}{K; \Gamma \vdash \textbf{match } r \textbf{ with } \{\text{`}tag_k(x_k) \Rightarrow r_k\}_1^n : \psi}\ (\textbf{Kmatch})$$

$$\frac{K; \Gamma \vdash r : \sigma \quad (\rho \notin ftv(\Gamma))}{K; \Gamma \vdash r : \forall \rho . \sigma}\ (\textbf{K}\forall\rho) \qquad \frac{K; \Gamma \vdash r : \forall \rho . \sigma \quad K; \Gamma \vdash s : T}{K; \Gamma \vdash rs : \sigma[\rho := T]}\ (\textbf{Kinst}\rho)$$

$$\frac{K \oplus i_{\geq \langle L,U \rangle}; \Gamma \vdash r : \sigma}{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma}\ (\textbf{K}\forall\textbf{i}) \qquad \frac{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle} . \sigma \quad K \vdash i' \geq \langle L, U \rangle}{K; \Gamma \vdash r : \sigma[i := i']}\ (\textbf{Kinsti})$$

Bounded polymorphic variants

We assume the types of tags to be fixed in advance in a global

$$S = tag_1 : \phi_{tag_1}, \ldots, tag_n : \phi_{tag_n}$$

No separate compilation, adding a new tag will require re-compilation without code refactoring.

Syntax of types:

$$\begin{array}{rcl} T & ::= & \ldots \mid [i] \mid \forall i_{\geq \langle L,U \rangle}.T \\ L, U & ::= & \emptyset \mid i, L \\ K & ::= & \emptyset \mid K \oplus i_{\geq \langle L,U \rangle} \end{array}$$

All types $[j]$ are just subtypes of $[i]$ for $i_{\geq \langle \emptyset, \{tag_i\} \rangle}$.

Bounded polymorphic variants

$$\frac{(L \subseteq L' \quad H \subseteq H')}{K \oplus i_{\geq \langle L,U \rangle} \vdash i \geq \langle L', H' \rangle}\ (\textbf{Ki}) \qquad \frac{K; \Gamma \vdash r : \phi_{tag} \quad K \vdash i \geq \langle tag, \top \rangle}{K; \Gamma \vdash \text{`}tag(r) : [i]}\ (\textbf{Kvariant})$$

$$\frac{K; \Gamma \vdash r : [i] \quad K \vdash i \geq \langle \bot, \{tag_k\}_1^n \rangle \quad K; \Gamma, x_k : \phi_{tag_k} \vdash r_k : \psi \quad (1 \leq k \leq n)}{K; \Gamma \vdash \textbf{match } r \textbf{ with } \{\text{`}tag_k(x_k) \Rightarrow r_k\}_1^n : \psi}\ (\textbf{Kmatch})$$

$$\frac{K \oplus i_{\geq \langle L,U \rangle}; \Gamma \vdash r : \sigma}{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle}.\sigma}\ (\textbf{K}\forall\textbf{i}) \qquad \frac{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle}.\sigma \quad K \vdash i' \geq \langle L, U \rangle}{K; \Gamma \vdash r : \sigma[i := i']}\ (\textbf{Kinsti})$$

Bounded polymorphic variants in type theory

In a preamble file:

1. The algebraic type that declares every constructor

   inductive Expr (T) :=
   Const: $\mathbb{N} \to$ Expr T
   | Plus: T $\to$ T $\to$ Expr T
   | Mult: T $\to$ T $\to$ Expr

2. An enumeration of tags (names) for the constructors

   inductive tag := CONST | PLUS | MULT

3. Computable equality test

   definition eq_tags : tag $\to$ tag $\to$ $\mathbb{B}$

4. Classifier

   definition tag_of_expr : $\forall$ T. Expr T $\to$ tag

Bounded polymorphic variants in type theory

1. A function to compute the type of branches during pattern matching

```
definition Q_for_tag: ∀ T. ∀Q : Expr T → Prop. tag → Prop :=
 λT,Q,tag.
  match tag with
      CONST ⇒ ∀n. Q(Const n)
    | PLUS ⇒ ∀ E1,E2. Q(Plus E1 E2)
    | MULT ⇒ ∀ E1,E2. Q(Mult E1 E2)
```

2. A proof of exhaustivity of pattern matching

```
theorem Q_holds_for_tag: ∀ T. ∀Q: Expr T →Prop.
 ∀x: Expr T. Q_for_tag T Q (tag_of_expr T x) → Q x.
```

Bounded polymorphic variants in type theory

The encoding :

$$
\begin{array}{rcl}
[\![\forall i_{\geq\langle L,U\rangle}.T]\!] & = & \bigcap_{i:\text{list tag}}.L \subseteq i \wedge i \subseteq U.[\![T]\!] \\
[\![[i]]\!] & = & \Sigma x : \text{Expr } T.\text{tag\_of\_expr} x \in i \\
[\![`tag_i(t)]\!] & = & \langle tag_i([\![t]\!]), p\rangle \\
[\![\begin{array}{l}\text{match } x \text{ with} \\ \{`tag_i(x) => t_i\}\end{array} ]\!] & = & \begin{array}{l}\text{match } x \text{ with} \\ \{`tag_i(x) => \lambda H : \text{tag\_of\_expr } x \in i.[\![t_i]\!] \\ \quad \_ => \lambda H : \text{tag\_of\_expr } x \in i.\bot_e(p)\}\end{array} \\
\text{induction} & = & \text{application of Q\_holds\_for\_tag}
\end{array}
$$

Optimization: if $L = [tag_1; \ldots; tag_n]$ then

$$
\begin{array}{rcl}
[\![\forall i_{\geq\langle L,U\rangle}.T]\!] & = & \bigcap_{j:\text{list tag}}.\ \text{let } i := L@j \text{ in } i \subseteq U.[\![T]\!] \\
[\![`tag_i(t)]\!] & = & \langle tag_i([\![t]\!]), \star\rangle
\end{array}
$$

Bounded polymorphic variants in type theory

The encoding after code extraction:

$$
\begin{aligned}
[\![\forall i_{\geq\langle L,U\rangle}.T]\!] &= [\![T]\!] \\
[\![[i]]\!] &= \text{Expr } T \\
[\![`tag_i(t)]\!] &= tag_i([\![t]\!]) \\
[\![\ \text{match } x \text{ with } \{`tag_i(x) => t_i\}\ ]\!] &= \begin{array}{l}\text{match } x \text{ with} \\ \{`tag_i(x) => [\![t_i]\!] \\ \quad \_ => \bot_e\}\end{array}
\end{aligned}
$$

Subtyping

Slogan: "*structural subtyping as instantiation*"

$$\frac{K; \Gamma \vdash r : \forall i_{\geq \langle L,U \rangle}.\sigma \quad K \vdash i' \geq \langle L, U \rangle}{K; \Gamma \vdash r : \sigma[i := i']} \textbf{(Kinsti)}$$

In the encoding:

$$\frac{r : [\![\forall i_{\geq \langle L,U \rangle}.\sigma]\!] \quad p : L \subseteq i' \subseteq U}{r\ i'\ p : [\![\sigma]\!][i := i']} \textbf{(Kinsti)}$$

But also: explicit structural subtyping via coercions.

definition coerce: $\forall\, l_1, l_2$: list tag. $l_1 \subseteq l_2 \rightarrow [\![[l_1]]\!] \rightarrow [\![[l_2]]\!]$

(extracts to identity)

Outline

Conclusions

A lightweight encoding of bounded polymorphic variants in type theory

1. in user space only
2. adding a tag requires a change to the global preamble and recompilation, but no refactoring
3. maximally efficient code after code extraction
4. the proof obligations are handled by trivial automation
5. heavily used in the EU project CerCo to formalize the 8051 microprocessor (almost every opcode takes a different subset of the allowed operand modes)

Bounded extensible records

The encoding is less satisfactory:

record status := $\{$ field$_1$: option T_1; . . .; field$_n$: option T_n $\}$

$[\![\{i\}]\!] = \Sigma s$: status.all_fields_set i s

Related work

Polymorphic variants not in Haskell: explicit use of functors and fixpoints in “datatypes à la carte”

Can be made to work with dependent types too
(see Matteo Acerbi’s Master Thesis, based on indexed descriptions, implemented in Agda)

Way more cumbersome for the user, slightly less efficient, requires code refactoring or quotienting (??).