DEPARTMENT OF COMPUTER & INFORMATION SCIENCES

University of Strathclyde Science

Generic Level Polymorphic N-ary Functions

Guillaume ALLAIS

SPLS @ LFCS

Faculty of Science

June 17, 2019

State Of the Art
- N-ary Combinators... for N up to 2
- Working with Indexed Families

Requirements

Getting Acquainted With the Unifier

Generic Level Polymorphic N-ary Functions
- Unification-Friendly Representation
- N-ary Combinators

Going Further

State Of the Art : N-ary Combinators... for N up to 2

Propositional equality as an inductive family:

```
data _≡_ {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

Congruence and substitution proven by pattern-matching:

```
cong : (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

```
subst : (P : A → Set p) → x ≡ y → P x → P y
subst P refl px = px
```

University of Strathclyde Science

State Of the Art : N-ary Combinators... for N up to 2

Binary Versions

Binary congruence and substitution proven by pattern-matching:

$$\begin{aligned}
&\mathsf{cong}_2 : (f : A \to B \to C) \to \\
&\qquad x \equiv y \to t \equiv u \to f\,x\,t \equiv f\,y\,u \\
&\mathsf{cong}_2\ f\ \mathsf{refl}\ \mathsf{refl} = \mathsf{refl}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{subst}_2 : (R : A \to B \to \mathsf{Set}\ p) \to \\
&\qquad x \equiv y \to t \equiv u \to R\,x\,t \to R\,y\,u \\
&\mathsf{subst}_2\ P\ \mathsf{refl}\ \mathsf{refl}\ pr = pr
\end{aligned}$$

State Of the Art : N-ary Combinators... for N up to 2

What we would like to have: n-ary congruence and substitution.

$$
\begin{aligned}
\mathsf{cong}_n : & (f : A_1 \to \cdots \to A_n \to B) \to \\
& a_1 \equiv b_1 \to \cdots \to a_n \equiv b_n \to \\
& f\, a_1\, \cdots\, a_n \equiv f\, b_1\, \cdots\, b_n
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{subst}_n : & (R : A_1 \to \cdots \to A_n \to \mathsf{Set}\ r) \to \\
& a_1 \equiv b_1 \to \cdots \to a_n \equiv b_n \to \\
& R\, a_1\, \cdots\, a_n \to R\, b_1\, \cdots\, b_n
\end{aligned}
$$

University of Strathclyde Science

State Of the Art : Working with Indexed Families

Example datatype our families will be indexed over:

```
data List (A : Set a) : Set a where
  []   : List A
  _::_ : A → List A → List A
```

Predicate transformer: *P* holds of all the values in the list:

```
data All (P : A → Set p) : List A → Set (a ⊔ p) where
  []   : All P []
  _::_ : P x → All P xs → All P (x :: xs)
```

State Of the Art : Working with Indexed Families

Explicit and implicit universal quantifier:

$$\Pi[\_] : (I \to \text{Set } p) \to \text{Set } (i \sqcup p)$$
$$\Pi[\ P\ ] = \forall\ i \to P\ i$$

$$\forall[\_] : (I \to \text{Set } p) \to \text{Set } (i \sqcup p)$$
$$\forall[\ P\ ] = \forall\ \{i\} \to P\ i$$

Example: if P is universally true, then it holds of all the elements of any list.

$$\text{replicate} : \forall[\ P\ ] \to \Pi[\ \text{All } P\ ]$$
$$\text{replicate } p\ [] \qquad = []$$
$$\text{replicate } p\ (x :: xs) = p :: \text{replicate } p\ xs$$

State Of the Art : Working with Indexed Families

Lifting implication between Sets to implication between families:

$$\_\Rightarrow\_ : (I \to \mathsf{Set}\ p) \to (I \to \mathsf{Set}\ q) \to (I \to \mathsf{Set}\ (p \sqcup q))$$
$$(P \Rightarrow Q)\ i = P\ i \to Q\ i$$

Example: Applicative's 'ap' for All:

$$\_{<}{\star}{>}\_ : \forall[\ \mathsf{All}\ (P \Rightarrow Q) \Rightarrow \mathsf{All}\ P \Rightarrow \mathsf{All}\ Q\ ]$$
$$[]\ {<}{\star}{>}\ [] = []$$
$$(f :: fs)\ {<}{\star}{>}\ (x :: xs) = f\ x :: (fs\ {<}{\star}{>}\ xs)$$

University of Strathclyde Science

State Of the Art : Working with Indexed Families

Adjustments To The Ambient Index

Updating the index we are talking about:

$$\_\vdash\_ : (I \to J) \to (J \to \mathsf{Set}\ p) \to (I \to \mathsf{Set}\ p)$$
$$(f \vdash P)\ i = P\ (f\ i)$$

Example: concat's action on the predicate transformer All:

$$\mathsf{concat}^+ : \forall[\ \mathsf{All}\ (\mathsf{All}\ P) \Rightarrow \mathsf{concat} \vdash \mathsf{All}\ P\ ]$$
$$\mathsf{concat}^+\ [] = []$$
$$\mathsf{concat}^+\ ([]\ ::\ pxss) = \mathsf{concat}^+\ pxss$$
$$\mathsf{concat}^+\ ((px :: pxs) :: pxss) = px :: \mathsf{concat}^+\ (pxs :: pxss)$$

Requirements

Wishes

1. Reified types of n-ary functions (including level polymorphism)
2. Semantics which should be
   - computable (including its Set-level)
   - invertible (to minimise user input)
3. Applications: generic programs
   - congruence, substitution
   - combinators for n-ary indexed families

Getting Acquainted With the Unifier

- Use case

  Mechanical process to reconstruct missing values:
  - Implicit arguments
  - Boring details the programmer left out

  Principled: the generated solutions (if any) are unique.

- Unification Problems: $lhs \approx rhs$
  - $?a$ stands for a metavariable
  - $e[?a_1, \cdots, ?a_n]$ for expression e mentioning $?a_1$ to $?a_n$
  - $c\ e_1 \cdots e_n$ for a constructor c applied to n expressions

University of Strathclyde Science

Getting Acquainted With the Unifier

Agda does unification all the time.

It is easy for us to ask Agda to solve unification problems

- Leave out values to create metavariables
- State that two expressions are equal to start a unification problem

```
_ : _
_ = _
```

For instance, $(?A \to ?B) \approx (\mathbb{N} \to \mathbb{N})$ and $(?A \to ?A) \approx (\mathbb{N} \to \mathbb{N})$

```
_ : (_ → _) ≡ (ℕ → ℕ)
_ = refl
```

```
_ : let ?A = _ in (?A → ?A) ≡ (ℕ → ℕ)
_ = refl
```

University of Strathclyde Science

Getting Acquainted With the Unifier

Problem: $?a \approx e\,[\,?a_1 \cdots ?a_n]$

Unifying a meta-variable with an expression.

1. Make sure $?a$ does not appear in $?a_1, \cdots, ?a_n$
2. Instantiate $?a$ to $e\,[\,?a_1 \cdots ?a_n]$
3. Discard the problem

Example:

```
_ : _ ≡ (_ → _)
_ = refl
```

University of Strathclyde Science

Getting Acquainted With the Unifier

Problem: $c\ e_1\ \cdots\ e_m \approx d\ f_1\ \cdots\ f_n$

Unifying two constructor-headed expressions.

1. Make sure the constructors *c* and *d* are equal
2. This means *m* equals *n*
3. Replace problem with subproblems $(e_1 \approx f_1) \cdots (e_m \approx f_n)$

Example:

$$\_ : (\mathbb{N} \to \_) \equiv (\mathbb{N} \to \mathbb{N})$$
$$\_ = \text{refl}$$

University of Strathclyde Science

Getting Acquainted With the Unifier

Avoid generating unification problems involving recursive functions.

$$\begin{array}{l} \mathsf{nary} : \mathbb{N} \to \mathsf{Set} \to \mathsf{Set} \\ \mathsf{nary}\ \mathsf{zero}\ \ \ \ \ \ \ A = A \\ \mathsf{nary}\ (\mathsf{suc}\ n)\ A = \mathbb{N} \to \mathsf{nary}\ n\ A \end{array}$$

$$\begin{array}{l} \_ : \mathsf{nary}\ \_\ \_ \equiv (\mathbb{N} \to \mathbb{N}) \\ \_ = \mathsf{refl} \end{array}$$

Getting Acquainted With the Unifier

Avoid generating unification problems involving recursive functions.

$$\begin{array}{l} \text{nary} : \mathbb{N} \to \text{Set} \to \text{Set} \\ \text{nary zero} \quad A = A \\ \text{nary (suc } n) \; A = \mathbb{N} \to \text{nary } n \; A \end{array}$$

$$\begin{array}{l} \_ : \text{nary} \; \_ \; \_ \equiv (\mathbb{N} \to \mathbb{N}) \\ \_ = \text{refl} \end{array}$$

Unless the recursion goes away in the cases you are interested in.

$$\begin{array}{l} \_ : \text{nary } 0 \; \_ \equiv (\mathbb{N} \to \mathbb{N}) \\ \_ = \text{refl} \end{array}$$

$$\begin{array}{l} \_ : \text{nary } 1 \; \_ \equiv (\mathbb{N} \to \mathbb{N}) \\ \_ = \text{refl} \end{array}$$

Getting Acquainted With the Unifier

Avoid Computations... Unless (Part II)

Avoid generating unification problems involving recursive functions.

$$\begin{array}{l} \text{nary} : \mathbb{N} \to \text{Set} \to \text{Set} \\ \text{nary zero} \quad A = A \\ \text{nary (suc } n\text{)}\ A = \mathbb{N} \to \text{nary}\ n\ A \end{array}$$

$$\begin{array}{l} \_ : \text{nary}\ \_\ (\mathbb{N} \to \mathbb{N}) \equiv (\mathbb{N} \to \mathbb{N}) \\ \_ = \text{refl} \end{array}$$

Getting Acquainted With the Unifier

Avoid Computations... Unless (Part II)

Avoid generating unification problems involving recursive functions.

$$
\begin{aligned}
&\mathsf{nary} : \mathbb{N} \to \mathsf{Set} \to \mathsf{Set}\\
&\mathsf{nary}\ \mathsf{zero}\quad A = A\\
&\mathsf{nary}\ (\mathsf{suc}\ n)\ A = \mathbb{N} \to \mathsf{nary}\ n\ A
\end{aligned}
\qquad
\begin{aligned}
&\_ : \mathsf{nary}\ \_\ (\mathbb{N} \to \mathbb{N}) \equiv (\mathbb{N} \to \mathbb{N})\\
&\_ = \mathsf{refl}
\end{aligned}
$$

Unless the recursion is trivially invertible.

$$
\begin{aligned}
&\_ : \mathsf{nary}\ \_\ \mathbb{N} \equiv \mathbb{N}\\
&\_ = \mathsf{refl}
\end{aligned}
\qquad
\begin{aligned}
&\_ : \mathsf{nary}\ \_\ \mathbb{N} \equiv (\mathbb{N} \to \mathbb{N})\\
&\_ = \mathsf{refl}
\end{aligned}
$$

Generic Level Polymorphic N-ary Functions

We want to

- Define representation of *n*-ary functions
- Give it a semantics (here called $[\![\_]\!]$)

Such that when faced with constraints involving concrete types, Agda can easily reconstruct the representation.

Example: recover *?r* from $[\![\ ?r\ ]\!] \approx (\mathbb{N} \to \mathsf{Set})$

Generic Level Polymorphic N-ary Functions : Unification-Friendly Representation

Representation

```
Levels : ℕ → Set
Levels zero    = ⊤
Levels (suc n) = Level × Levels n
```

Generic Level Polymorphic N-ary Functions : Unification-Friendly Representation

Representation

$$
\begin{aligned}
&\mathsf{Levels} : \mathbb{N} \to \mathsf{Set}\\
&\mathsf{Levels}\ \mathsf{zero} \quad = \top\\
&\mathsf{Levels}\ (\mathsf{suc}\ n) = \mathsf{Level} \times \mathsf{Levels}\ n
\end{aligned}
$$

$$
\begin{aligned}
&\bigsqcup : \forall\ n \to \mathsf{Levels}\ n \to \mathsf{Level}\\
&\bigsqcup\ \mathsf{zero} \quad \_ \quad = 0\ell\\
&\bigsqcup\ (\mathsf{suc}\ n)\ (l\ ,\ ls) = l \sqcup (\bigsqcup\ n\ ls)
\end{aligned}
$$

Generic Level Polymorphic N-ary Functions : Unification-Friendly Representation

Representation

$$\begin{aligned}
&\mathsf{Levels} : \mathbb{N} \to \mathsf{Set}\\
&\mathsf{Levels}\ \mathsf{zero} = \top\\
&\mathsf{Levels}\ (\mathsf{suc}\ n) = \mathsf{Level} \times \mathsf{Levels}\ n
\end{aligned}$$

$$\begin{aligned}
&\bigsqcup : \forall\ n \to \mathsf{Levels}\ n \to \mathsf{Level}\\
&\bigsqcup\ \mathsf{zero}\ _ = 0\ell\\
&\bigsqcup\ (\mathsf{suc}\ n)\ (l\ ,\ ls) = l \sqcup (\bigsqcup\ n\ ls)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Sets} : \forall\ n\ (ls : \mathsf{Levels}\ n) \to \mathsf{Set}\ (\mathsf{Level.suc}\ (\bigsqcup\ n\ ls))\\
&\mathsf{Sets}\ \mathsf{zero}\ _ = \mathsf{Lift}\ _\ \top\\
&\mathsf{Sets}\ (\mathsf{suc}\ n)\ (l\ ,\ ls) = \mathsf{Set}\ l \times \mathsf{Sets}\ n\ ls
\end{aligned}$$

Generic Level Polymorphic N-ary Functions : Unification-Friendly Representation

Representation

$$
\begin{aligned}
&\mathsf{Levels} : \mathbb{N} \to \mathsf{Set}\\
&\mathsf{Levels}\ \mathsf{zero} \quad\;\; = \top\\
&\mathsf{Levels}\ (\mathsf{suc}\ n) = \mathsf{Level} \times \mathsf{Levels}\ n
\end{aligned}
$$

$$
\begin{aligned}
&\bigsqcup : \forall\ n \to \mathsf{Levels}\ n \to \mathsf{Level}\\
&\bigsqcup\ \mathsf{zero} \quad\;\; \_ \qquad\;\; = 0\ell\\
&\bigsqcup\ (\mathsf{suc}\ n)\ (l\ ,\ ls) = l \sqcup (\bigsqcup\ n\ ls)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Sets} : \forall\ n\ (ls : \mathsf{Levels}\ n) \to \mathsf{Set}\ (\mathsf{Level.suc}\ (\bigsqcup\ n\ ls))\\
&\mathsf{Sets}\ \mathsf{zero} \quad\;\; \_ \qquad\;\; = \mathsf{Lift}\ \_\ \top\\
&\mathsf{Sets}\ (\mathsf{suc}\ n)\ (l\ ,\ ls) = \mathsf{Set}\ l \times \mathsf{Sets}\ n\ ls
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Arrows} : \forall\ n\ \{ls\} \to \mathsf{Sets}\ n\ ls \to \mathsf{Set}\ r \to \mathsf{Set}\ (r \sqcup (\bigsqcup\ n\ ls))\\
&\mathsf{Arrows}\ \mathsf{zero} \quad\;\; \_ \qquad\;\; b = b\\
&\mathsf{Arrows}\ (\mathsf{suc}\ n)\ (a\ ,\ as)\ b = a \to \mathsf{Arrows}\ n\ as\ b
\end{aligned}
$$

Generic Level Polymorphic N-ary Functions : N-ary Combinators

Congruence

$$\mathsf{Cong}_n : \forall\ n\ \{ls\}\ \{as : \mathsf{Sets}\ n\ ls\}\ \{R : \mathsf{Set}\ r\} \to$$
$$(f\ g : \mathsf{Arrows}\ n\ as\ R) \to \mathsf{Set}\ (r \sqcup (\bigsqcup\ n\ ls))$$
$$\mathsf{Cong}_n\ \mathsf{zero}\ \ f\ g = f \equiv g$$
$$\mathsf{Cong}_n\ (\mathsf{suc}\ n)\ f\ g = \forall\ \{x\ y\} \to x \equiv y \to \mathsf{Cong}_n\ n\ (f\ x)\ (g\ y)$$

$$\mathsf{cong}_n : \forall\ n\ \{ls\}\ \{as : \mathsf{Sets}\ n\ ls\}\ \{R : \mathsf{Set}\ r\} \to$$
$$(f : \mathsf{Arrows}\ n\ as\ R) \to \mathsf{Cong}_n\ n\ f\ f$$
$$\mathsf{cong}_n\ \mathsf{zero}\ \ f\ = \mathsf{refl}$$
$$\mathsf{cong}_n\ (\mathsf{suc}\ n)\ f\ \mathsf{refl} = \mathsf{cong}_n\ n\ (f\ \_)$$

Generic Level Polymorphic N-ary Functions : N-ary Combinators

Lifting of Type Constructors

$$
\begin{aligned}
&\mathsf{lift}_2 : \forall\ n\ \{ls\}\ \{as : \mathsf{Sets}\ n\ ls\} \to (A \to B \to C) \to \\
&\qquad \mathsf{Arrows}\ n\ as\ A \to \mathsf{Arrows}\ n\ as\ B \to \mathsf{Arrows}\ n\ as\ C \\
&\mathsf{lift}_2\ \mathsf{zero}\qquad op\ f\ g = op\ f\ g \\
&\mathsf{lift}_2\ (\mathsf{suc}\ n)\ op\ f\ g = \lambda\ x \to \mathsf{lift}_2\ n\ op\ (f\ x)\ (g\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\_\Rightarrow\_ : \mathsf{Arrows}\ n\ \{ls\}\ as\ (\mathsf{Set}\ r) \to \mathsf{Arrows}\ n\ as\ (\mathsf{Set}\ s) \to \\
&\qquad \mathsf{Arrows}\ n\ as\ (\mathsf{Set}\ (r \sqcup s)) \\
&\_\Rightarrow\_ = \mathsf{lift}_2\ \_\ (\lambda\ A\ B \to (A \to B))
\end{aligned}
$$

University of Strathclyde Science

Generic Level Polymorphic N-ary Functions : N-ary Combinators

Adjustments To The Ambient Index

$$\_\%\!=\_\vdash\_ : \forall\ n\ \{ls\}\ \{as : \mathsf{Sets}\ n\ ls\} \to (I \to J) \to$$
$$\qquad \mathsf{Arrows}\ n\ as\ (J \to B) \to \mathsf{Arrows}\ n\ as\ (I \to B)$$
$$\mathsf{zero}\ \ \%\!=\ f \vdash g = g \circ f$$
$$\mathsf{suc}\ n\ \%\!=\ f \vdash g = (n\ \%\!=\ f \vdash \_) \circ g$$

Going Further

Draft: `https://gallais.github.io/pdf/tyde19_draft.pdf`

- Already merged in the standard library:
  - Unification-friendly representation of n-ary functions and products
  - Proofs of n-ary congruence and substitution
  - Combinators for n-ary relations and functions
  - Direct style printf
- Coming up:
  - n-ary version of zipWith & friends
- Future work:
  - Dependent n-ary functions and products

Appendix


```
data Chunk : Set where
  Nat  : Chunk
  Raw  : String → Chunk

Format : Set
Format = List Chunk

format : (fmt : Format) → Sets (size fmt) 0ℓs
format []            = _
format (Nat   :: f)  = ℕ , format f
format (Raw _ :: f)  = format f

assemble : ∀ fmt → Product _ (format fmt) → List String
assemble []             vs       = []
assemble (Nat   :: fmt) (n , vs) = show n :: assemble fmt vs
assemble (Raw s :: fmt) vs       = s :: assemble fmt vs

printf : ∀ fmt → Arrows _ (format fmt) String
printf fmt = curryₙ (size fmt) (concat ∘ assemble fmt)
```