Generic Level Polymorphic N-ary Functions

Guillaume Allais
University of Strathclyde
Glasgow, UK
guillaume.allais@strath.ac.uk

Abstract

Agda's standard library struggles in various places with n-ary functions and relations. It introduces congruence and substitution operators for functions of arities one and two, and provides users with convenient combinators for manipulating indexed families of arity exactly one.

After a careful analysis of the kinds of problems the unifier can easily solve, we design a unifier-friendly representation of n-ary functions. This allows us to write generic programs acting on n-ary functions which automatically reconstruct the representation of their inputs' types by unification. In particular, we can define fully level polymorphic n-ary versions of congruence, substitution and the combinators for indexed families, all requiring minimal user input.

ACM Reference Format:
Guillaume Allais. 2019. Generic Level Polymorphic N-ary Functions. In *Proceedings of DRAFT (DRAFT'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.475/123_4

Introduction

For user convenience, Agda's standard library has accumulated a set of equality-manipulating combinators of varying arities (Section 1) as well as a type-level compositional DSL to write clean types involving indexed families of arity exactly one (Section 2.1). None of these solutions scale well. By getting acquainted with the unifier (Section 4), we can design a good representation of n-ary function spaces (Section 5) which empowers us to write generalised combinators (Sections 6 and 7) usable with minimal user input. We then see how the notions introduced to tackle our original motivations can be mobilised for other efforts in generic programming from an arity-generic zipWith (Section 8.2) to a direct style definition of printf (Section 8.3). All the code in this paper is written in Agda [12].

DRAFT'19, May 2019,

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

1 N-ary Combinators... for N up to 2

Agda's standard library relies on propositional equality defined as a level polymorphic inductive family with one constructor refl.

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

As one would expect from a notion of equality, it is congruent (i.e. for any function equal inputs yield equal outputs) and substitutive (i.e. equals behave the same with respect to predicates). Concretely this means we can write the two following functions by dependent pattern-matching on the equality proof:

```
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
```

However we quickly realise that it is convenient to be able to use congruence for functions which take more than one argument and substitution for at least binary relations. The standard library provides binary versions of both of these functions:

```
cong₂ : (f : A → B → C) →
        x ≡ y → t ≡ u → f x t ≡ f y u
cong₂ f refl refl = refl

subst₂ : (R : A → B → Set p) →
         x ≡ y → t ≡ u → R x t → R y u
subst₂ P refl refl pr = pr
```

If we want to go beyond arity two we are left to either define our own ternary, quaternary, etc. versions of cong and subst or awkwardly chaining the ones with a lower arity to slowly massage the expression at hand into the shape we want. Both of these solutions are unsatisfactory.

Wish We would like to define once and for all two functions cong_n and subst_n of respective types (pseudocode):

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

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

2 Invariant Respecting Programs

A key feature of dependently typed languages is the ability to enforce strong invariants. Inductive families [5] are essentially classic inductive types where one may additionally bake in these strong invariants. As soon as the programmer starts making these constraints explicit, they need to write constraints-respecting programs. Although a lot of programs are index-preserving, users need to be painfully explicit about things that stay the same (i.e. the index being threaded all across the function's type) rather than being able to highlight the important changes.

2.1 Working With Indexed Families

The standard library defines a set of handy combinators to talk about indexed families without having to manipulate their index explicitly. These form a compositional type-level Domain Specific Language [8] (DSL): each combinator has a precise semantics and putting them together builds an overall meaning.

A typical expression built using this DSL follows a fairly simple schema: a combinator acting as a quantifier for the index (Section 2.1.1) surrounds a combination of pointwise liftings of common type constructors (Section 2.1.2), index updates (Section 2.1.3), and base predicates. This empowers us to write lighter types which hide away the bits that are constant, focusing instead on the key predicates and the changes made to the index.

Before we can even talk about concrete indexed families, describe these various combinators, and demonstrate their usefulness, we need to introduce the data the families in our running examples will be indexed over. We pick List the level polymorphic type of lists parameterised by the type of their elements: they are both well-known and complex enough to allow us to write interesting types.

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

The most straightforward non-trivial indexed family we can define over List is the predicate lifting All which ensures that a given predicate P holds of all the elements of a list. It has two constructors which each bear the same name as their counterparts in the underlying data: nil ([]) is a proof that all the elements in the empty list satisfy P and cons (_::_) is a proof that P holds of all the elements of a non-empty list if it holds of its head and of all the elements in its tail.

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

Some of our examples require the introduction of Any, the other classic predicate lifting on list. It takes a predicate and ensures that it holds of at least one element of the list at hand. Either it holds of the first one and we are given a proof (here) or it holds of a value somewhere in the tail (there).

```
data Any (P : A → Set p) : List A → Set (a ⊔ p) where
  here  : P x → Any P (x :: xs)
  there : Any P xs → Any P (x :: xs)
```

2.1.1 Quantifiers

We have two types of quantifiers: existential and universal. As they are meant to *surround* the indexed expression they are acting upon, we define them as essentially pairs of matching opening and closing brackets. The opening one is systematically decorated with a mnemonic symbol: ∃ for existential quantification, Π for explicit dependent quantification and ∀ for implicit universal quantification. Additionally we use chevrons for existential quantifiers and square brackets for universal ones, recalling the operators diamond and box of modal logic.

Existential Quantifier In type theory, existential quantifiers are represented as dependent pairs. We introduce Σ, a dependent record parameterised by a type A and a type family P. It has two fields proj_1 for a value of type A and proj_2 for a proof of type $(P\ \text{proj}_1)$. We can build and pattern-match against pairs using the constructor _,_ and we can project either of the pair's components simply by using its field's name.

```
record Σ (A : Set a) (P : A → Set p) : Set (a ⊔ p) where
  constructor _,_
  field proj₁ : A
        proj₂ : P proj₁
```

The existential quantifier for indexed families is defined as a special case of Σ which takes the index Set implicitly.

```
∃⟨_⟩ : {A : Set a} (P : A → Set p) → Set (a ⊔ p)
∃⟨ P ⟩ = Σ _ P
```

Using ∃⟨_⟩ we can write our first statement about an indexed family: from the existence of a list such that P holds of all its elements, we can construct a list of pairs of elements and proofs that P holds for that value.

```
toList : ∃⟨ All P ⟩ → List ∃⟨ P ⟩
toList ([]       , [])       = []
toList (x :: xs , p :: ps) = (x , p) :: toList (xs , ps)
```

Universal Quantifiers The pendant of existential quantification is universal quantification. In type theory this corresponds to a dependent function space. Here we have room for variations and we can consider both the explicit (Π[_]) *and* the implicit (∀[_]) universal quantifiers.

```
Π[_] : (I → Set p) → Set (i ⊔ p)
Π[ P ] = ∀ i → P i
```

```
∀[_] : (I → Set p) → Set (i ⊔ p)
∀[ P ] = ∀ {i} → P i
```

Provided that a proposition holds of any value, we can prove it will hold of any list of values by induction on such a list. Because we perform induction on the list it is convenient to take it as an explicit argument whereas the proof itself can take its argument implicitly.

```
replicate : ∀[ P ] → Π[ All P ]
replicate p []       = []
replicate p (x :: xs) = p :: replicate p xs
```

2.1.2 Pointwise Liftings

Pointwise liftings for an index type I are operators turning a type constructor on Sets into one acting on I-indexed families by threading the index. They are meant to be used partially applied so that both their inputs and their output are I-indexed, hence the mismatch between their arity and the number of places for their arguments.

Implication We start with the most used of all: implication i.e. functions from proofs of one predicate to proofs of another.

```
_⇒_ : (I → Set p) → (I → Set q) → (I → Set (p ⊔ q))
(P ⇒ Q) i = P i → Q i
```

The combinator _⇒_ associates to the right just like the type constructor for functions does. We can write the analogue of sequential application for applicative functors [11] like so:

```
_<★>_ : ∀[ All (P ⇒ Q) ⇒ All P ⇒ All Q ]
[]       <★> []       = []
(f :: fs) <★> (x :: xs) = f x :: (fs <★> xs)
```

Conjunction To state that the conjunction of two predicates hold we can use the pointwise lifting of pairing.

```
_∩_ : (I → Set p) → (I → Set q) → (I → Set (p ⊔ q))
(P ∩ Q) i = Σ (P i) λ _ → Q i
```

This enables us to write functions which return more than one result. We can for instance write the type of unzip, the proof that if the conjunction of P and Q holds of all the elements of a given list then both P and Q in isolation hold of all of that list's elements.

```
unzip : ∀[ All (P ∩ Q) ⇒ All P ∩ All Q ]
unzip []             = [] , []
unzip ((p , q) :: pqs) = let (ps , qs) = unzip pqs
                         in (p :: ps) , (q :: qs)
```

Notice that we are using the conjunction combinator both on predicates ranging over values and on ones ranging over lists of values.

Disjunction To formally describe the disjunction of two predicates, we need to define _⊎_ the type of disjoint sums first. It has two constructors each of which corresponds to a choice of one side of the sum or the other.

```
data _⊎_ (A : Set a) (B : Set b) : Set (a ⊔ b) where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

The disjunction of two predicates is then the pointwise lifting of _⊎_.

```
_∪_ : (I → Set p) → (I → Set q) → (I → Set (p ⊔ q))
(P ∪ Q) i = (P i) ⊎ (Q i)
```

A typical use case for disjoint sums is the notion of decidability: either a predicate or its negation holds. We can formulate a general decidability result for All: if for any value either P or Q holds then for any list of values, either (Any P) or (All Q) holds.

```
decide : Π[ P ∪ Q ] → Π[ Any P ∪ All Q ]
decide pq? []       = inj₂ []
decide pq? (x :: xs) with pq? x | decide pq? xs
... | inj₁ px | _      = inj₁ (here px)
... | _      | inj₁ ap = inj₁ (there ap)
... | inj₂ qx | inj₂ qxs = inj₂ (qx :: qxs)
```

Here we did not limit ourselves to either P or its negation but it is sometimes necessary to talk directly about negation.

Negation Traditionally negation is defined as functions into the empty type ⊥. We start by defining it as the inductive type with not constructor together with its elimination principle.

```
data ⊥ : Set where          ⊥-elim : ⊥ → A
                            ⊥-elim ()
```

Negation for a unary predicate P is then the unary predicate which maps i to $(P\ i \to \bot)$.

```
¬_ : (I → Set p) → (I → Set p)
(¬ P) i = P i → ⊥
```

The two predicate liftings All and Any interact in non-trivial ways. For instance if we know that the negation of P holds of any value in a given list then P can't hold of all its elements. In other words: a single counter-example is enough to disprove a universal statement.

```
notall : ∀[ Any (¬ P) ⇒ ¬ All P ]
notall (here ¬px) (px :: _) = ⊥-elim (¬px px)
notall (there ¬p) (_ :: ps) = notall ¬p ps
```

Notice that we are once more using the combinator we just defined both on a predicate on values and one on lists of values.

2.1.3 Adjustments To The Ambient Index

Threading the index is only the least invasive of the modes of action available to us. But we can also more actively interact with the ambient index either by ignoring it completely, adjusting it using a function or overwriting it entirely. We won't detail the last option as, as always, overwriting is adjusting with a constant function.

Constant Although we have so far only manipulated indexed families, some of our function's arguments or its result may not depend on the index. The constant indexed family is precisely what we need to represent these cases.

```
const : Set a → (I → Set a)
const A i = A
```

We can for instance prove that if the constantly false predicate (const ⊥) holds true of all the elements of a list then said list is the empty list. We use a section (i.e. a partially applied infix operator) of propositional equality to formulate that conclusion. In the proof we do not need to consider the _::_ case: Agda automatically detects that it is impossible.

```
empty : ∀[ All (const ⊥) ⇒ (List A ∋ []) ≡_ ]
empty [] = refl
```

Note that we had to add a type annotation to []: the type of the index of the predicate defined using const is an implicit polymorphic argument and so is the type of elements in List's nil constructor. Agda can infer that these two implicit arguments are equal but needs to be given enough information to figure out what it ought to be. In type theory, an identity function is a fine definition of a type annotation operator:

```
_∋_ : ∀ {a} (A : Set a) → A → A
A ∋ a = a
```

Update On the other end of the spectrum, we have operations which update the ambient index using an arbitrary function. The notation _⊢_ is inspired by the convention in type theory to consider that proofs in sequent calculus are written in an ambient context and that we may use a turnstile to describe the addition of newly-bound variables to this context (see e.g. Martin Löf's work [9]).

```
_⊢_ : (I → J) → (J → Set p) → (I → Set p)
(f ⊢ P) i = P (f i)
```

Stating that a function operating on lists is compatible with All is a typical use case of such a combinator. If the function at hand is called f then the convention in the standard library is to call such a proof f⁺ as it makes f appear in the conclusion. We pick concat (whose classic definition is left out) in this concrete example.

```
concat⁺ : ∀[ All (All P) ⇒ concat ⊢ All P ]
concat⁺ []           = []
concat⁺ ([]     :: pxss) = concat⁺ pxss
concat⁺ ((px :: pxs) :: pxss) = px :: concat⁺ (pxs :: pxss)
```

2.2 Working With Multiple Indices

We started by showing both the type and the implementation of each of our examples. Although convenient at first to build an understanding of which arguments are explicit and which ones are implicit, we are in the end only interested in the way combinators let us write types. In this section, we focus on the types and only the types of our examples.

The combinators presented earlier are all available in the standard library. As we have demonstrated, they work really well for unary predicates. Unfortunately they do not scale beyond that. Meaning that if we are manipulating binary relations for instance we have to explicitly introduce one of the indices and partially apply the relations in question before we can use our usual unary combinators. This leads to cluttered types which are not much better than their fully expanded counterparts.

Let us look at an example. We introduce Pw (for "pointwise") the relational equivalent of the predicate lifting All we have been as our running example so far. The inductive family Pw is parameterised by a relation R and ensures its two index list are compatible with R in a pointwise manner. If both lists are empty then they are trivially related ([]); otherwise we demand that their heads are related by R and their tails are related pointwise (_::_).

```
data Pw (R : A → B → Set r) :
        List A → List B → Set (a ⊔ b ⊔ r) where
  []  : Pw R [] []
  _::_ : R x y → Pw R xs ys → Pw R (x :: xs) (y :: ys)
```

To state the relational equivalent to All's _<⋆>_ using our unary combinators to manipulate predicates, we need to partially apply Pw to xs to make it a predicate as well as explicitly use a λ-abstraction to build the relation corresponding to the fact that R implies S.

```
_<⋆>_ : ∀[ Pw (λ x → R x ⇒ S x) xs ⇒
           Pw R xs ⇒ Pw S xs ]
```

Ideally we could have instead used binary version of the combinators for unary predicates we saw earlier and have simply written:

```
_<⋆>_ : ∀[ Pw (R ⇒ S) ⇒ Pw R ⇒ Pw S ]
```

We could duplicate the definitions for unary predicates and have equivalent combinators for binary relations however this will create two new issues. First, the day we need a library for ternary relations we will have triplicated the initial work. Second, we would have two sets of definitions with identical names meaning they cannot be both imported in the same module without clashing thus forcing users to manually disambiguate each use site.

Wish We would like to define once and for all n-ary quantifiers, pointwise lifting of common type constructors, and adjustment functions.

3 Plan

We can start to draw out the structure of our contribution now that we have a good idea of the current state of the art, its limitation, and the extension we want to see. Here are the key points we need to deliver:

Reified Types We need to come up with a representation of n-ary functions which is as general as possible: the domains should be allowed to be different types, even types defined at different levels.

Semantics We need to give a semantics taking a reified type and computing its meaning as a Set at some level which will also need to be computed.

Invertible The representation and its semantics should be unifier friendly. That is to say that if using a combinators yields a constraint of the form "this type should be the result of evaluating the representation of an n-ary function type" then Agda should be able to reconstruct the representation and discharge the constraints without any outside help.

Applications Lastly we need to deliver the two wishes we formulated earlier by actually implementing the n-ary versions of cong, subst, and the various combinators for manipulating indexed families.

4 Getting Acquainted With the Unifier

Unification is the process by which Agda reconstructs the values of the implicit arguments the user was allowed to leave out [2, 3]. It is one of the mechanisms bridging the gap between the source program which should be convenient for humans to read, write, and modify and the fully explicit terms in the internal syntax.

It is important to build a good understanding of the problems the unifier can easily solve to be able to write combinators usable with minimal user input. Indeed if we can anticipate that an argument can be reconstructed, we may as well make it implicit and let Agda do the work.

Notations We write $?a$ for a metavariable, $e[?a_1, \cdots, ?a_n]$ for an expression e containing exactly the metavariables $?a_1$ to $?a_n$, $c\ e_1 \cdots e_n$ for the constructor c applied to n expressions and $lhs \approx rhs$ to state a unification problem between two expressions lhs and rhs.

Unification Tests We can easily trigger the resolution of unification problems by writing unit tests in the source language. We can force Agda to introduce metavariables by using an underscore (_) as a placeholder for a subterm and use refl at the proof of a propositional equality to force it to unify the two expressions stated to be equal. For instance in the following test we force Agda to check that $(?A \to ?B) \approx (\mathbb{N} \to \mathbb{N})$.

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

Whenever Agda cannot solve a metavariable by unification it is highlighted in yellow like so: _. Whenever Agda cannot satisfy a unification constraint raised by the use of refl, it will also highlight it in yellow like so: refl.

Let us now look at the various scenarios in which it is easy for the unifier to decide whether a constraint is satisfiable.

4.1 Instantiation

The simplest case the unifier can encounter is a problem of the form $?a \approx e[?a_1 \cdots ?a_n]$ where $?a$ does not appear in the list $[?a_1, \cdots, ?a_n]$. The unifier can simply instantiate the metavariable to the candidate expression.

For instance in the following test you can see that neither the underscore on the left-hand side nor the refl constructor is highlighted in yellow. Meaning that the metavariable on the left was indeed solved (by instantiating it to the expression on the right-hand side) and that the constraint induced by the use of refl was thus satisfied. The problem itself is under-constrained so it is not surprising that the right-hand side lights up.

```
_ : _ ≡ (_ → _)
_ = refl
```

4.2 Constructor Headed

The second case where the unifier can easily make progress is a unification problem between to constructor-headed expression $c\ e_1 \cdots e_m \approx d\ f_1 \cdots f_n$.

Success Either the constructors c and d match up, we learn that m equals n and we can reduce the problem to unifying the constructors' respective arguments by forming the new unification problems $(e_1 \approx f_1) \cdots (e_m \approx f_n)$.

In the following example, Agda sees that both expressions have _→_ as their head constructor, proceeds to unify ℕ with itself on the one hand (which succeeds because both have the same head constructor and they do not have any arguments) and ℕ with $?A$ on the other (which succeeds by instantiation).

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

Failure Or c and d are distinct and we can immediately conclude that unification is impossible. We cannot write an expression in Agda demonstrating this case as it leads to a type error in the language. Trying to form the unification problem $\mathbb{N} \approx (\_ \to \_)$ would raise such an error because ℕ and _→_ are distinct head constructors.

4.3 Avoid Computations...

In general unification problems involving computations are undecidable. We can easily construct a total simulation function sim for Turing machines which takes in as arguments the code for an arbitrary program *prg* and a natural number *n* and returns 0 if and only if the program runs for exactly *n* steps before stopping and 1 otherwise. Forming a constraint like sim *prg* _ ≈ 0 is effectively asking whether the program *prg* terminates. It is clearly impossible to write a unifier solving all problems of this form.

4.4 ... In Most Cases

Although unification problems involving computations will in general fail to produce solutions, there are exceptions.

Disappearing Problem The first favourable case is a Lapalissade: stuck function applications which are guaranteed to go away in all cases of interest to us are never a problem. This is true whenever we know that in all use cases the concrete values at hand will allow evaluation to reveal enough constructors for unification to succeed.

To demonstrate this phenomenon we introduce a type nary of *n*-ary functions on natural numbers. It is parameterised by the return type of the *n*-ary function and defined by induction *n*.

```
nary : ℕ → Set → Set
nary zero    A = A
nary (suc n) A = ℕ → nary n A
```

In general, it is impossible to solve the unification constraint nary *?n ?A* ≈ (ℕ → *A*). If the natural number is not specified then nary is stuck. And there is no hope to solve this problem; indeed there are two solutions because every unary function is also a nullary symbol whose type is a function type. As explained earlier, Agda communicates to us this failure to solve the two metavariables passed to nary as arguments by highlighting them in yellow, refl is also highlighted as the source of the unification constraint which could not be satisfied.

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

If the natural number argument is however a concrete value then nary evaluates fully and Agda is able to reconstruct *A* by unification. In the following two examples we unify (ℕ → ℕ) with (ℕ → *?A*) on the one hand and *?A* on the other. Both unification problems succeed without any issue.

```
_ : nary 1 _ ≡ (ℕ → ℕ)
_ = refl
```

```
_ : nary 0 _ ≡ (ℕ → ℕ)
_ = refl
```

This observation is language independent. It will directly influence our encoding: we expect our users to only ever use our generic congruence combinator with concrete arities. A representation defined by induction on such a natural number would therefore work well with the unifier.

Invertible Problem The second case in which we may encounter unification problems involving stuck computations and still see Agda find a solution is more language dependent but just as principled. Whenever the stuck function is defined by a set of equations whose right-hand sides are clearly anti-unifiable, we can invert it.

For instance if the Set parameter to nary is known to be ℕ then the right-hand side of the first equation is ℕ and the second's one has the shape (ℕ → _). These two are clearly disjoint and so Agda can invert nary and figure out that the arity we left out is 1.

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

If we had passed (ℕ → ℕ) instead then the two right-hand sides would not have been obviously disjoint and Agda would have given up on trying to invert nary.

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

These two examples tell us that we can hope to leave out a function's arity entirely if we statically know its codomain and it has a shape clearly anti-unifiable with the right-hand sides of our semantics of reified function types. Note in particular that combinators acting on relations (cf. Section 2.1) are manipulating functions whose codomain is always of the shape (Set _) which is clearly disjoint from (_ → _). We ought to be able to define their *n*-ary counterparts without having to mention *n* explicitly.

5 Representing N-ary Function Types

Now that we understand how the unifier works, we can design a generic representation and its semantics (called ⟦_⟧ here for convenience) so that whenever we have a constraint of the form ⟦ *?r* ⟧ ≈ (ℕ → Set), it can easily lead to the reconstruction of the representation *?r*.

User Input As we have just seen, a binary function type (*A* → *B* → *C*) with codomain *C* can also be seen as a unary function type with codomain (*B* → *C*). As a consequence in the general case there is no hope to get Agda to reconstruct the representation we have in mind without passing it at least a little bit of information. The least we can do is tell Agda the arity of the function. From this single natural number we will compute the shape of the whole representation.

Unification As we have just seen, if we want Agda to reconstruct the representation from a unification constraint then our best hope is that the semantics function evaluates fully and simply disappears. This means in particular that it should not get stuck on a pattern-matching analysing the

representation. This can be achieved with certainty by constraining our representation to only be built up from things we either will not pattern-match against (e.g. Sets) or type constructors which enjoy η-equality (i.e. for which values can always be made to look like they are in canonical form). Just like the representation, the semantics will have to be computed entirely from the natural number corresponding to the function's arity.

From these two observations, we decide that our representation will be parameterised by a natural number which we will use to compute a number of right-nested products.

Right-Nested Products The two basic building blocks of right-nested products are a binary product _×_ and the unit type ⊤.

We obtain the binary product as the non-dependent special case of Σ we introduced in Section 2.1.1. We did not mention it at the time but record types in Agda enjoy η-rules. That is to say that any value p of type (Σ A P) is definitionally equal to (proj_1 p , proj_2 p).

The unit type is defined as a record with no field. Every value of type ⊤ is equal to the canonical value tt.

```
record ⊤ : Set where
  constructor tt
```

Even though ⊤ is defined as a Set, we will sometimes need to use it at a higher level. The usual solution is to manually lift it to the appropriate level. Because Lift is also a record, it will not get in the way of reconstruction.

```
record Lift ℓ (A : Set a) : Set (ℓ ⊔ a) where
  constructor lift
  field lower : A
```

Level Polymorphism To achieve fully general level polymorphism, we need all the domains of our function type to be potentially at different levels. Luckily the notion of Level in Agda is a primitive Set and we can thus manipulate them just like any other values. In particular we can define containers storing them. Our first definition called Levels defines an n-tuple of Levels by induction on n.

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

Heterogeneous Domains Before we can generate the big right-nested n-tuple packaging the function's domains, we need to compute the level at which it is going to live. The definition of Σ makes clear that the product of two types living respectively at level a and b sits at level (a ⊔ b) i.e. the least upper bound of a and b. We define ⨆ as the generalisation of the least upper bound operator to (Levels n) by induction on n.

```
⨆ : ∀ n → Levels n → Level
⨆ zero    _       = 0ℓ
⨆ (suc n) (l , ls) = l ⊔ (⨆ n ls)
```

Knowing that (Set a) sits at level (suc a), it is natural to declare that our n-tuple of sets defined at various Levels will be defined at the successor of the generalised least upper bound of these Levels.

```
Sets : ∀ n (ls : Levels n) → Set (Level.suc (⨆ n ls))
Sets zero    _       = Lift _ ⊤
Sets (suc n) (l , ls) = Set l × Sets n ls
```

We can now encode an n-ary function space as essentially a collection ls of (Levels n) together with a corresponding n-tuple of type (Sets n ls) for the domains, and a level r and a (Set r) for the codomain.

Semantics This encoding has a straightforward semantics by induction on n and case analysis on the (Sets n ls) argument. A zero-ary function type is simply the codomain whilst a (suc n)-ary one is a unary function type whose codomain is the n-ary function type obtained by induction hypothesis.

```
Arrows : ∀ n {ls} → Sets n ls → Set r → Set (r ⊔ (⨆ n ls))
Arrows zero    _       b = b
Arrows (suc n) (a , as) b = a → Arrows n as b
```

If we look carefully at this definition we can notice that the function Arrows may only ever get stuck if the natural number is not concrete. Even though we do take the Sets argument apart, it is a product type and thus enjoys η-rules. We have achieved the degree of unifier-friendliness we were aiming for.

Our first example is a 2-ary function: our favourite indexed family All. The last element of the telescope, a value whose type is a lifted version of the unit type, can be inferred by Agda so we leave it out.

```
_ : Arrows 2 ((A → Set p) , List A , _) (Set (p ⊔ a))
_ = All
```

6 Combinators for Indexed Families

Now that we have our generic representation of n-ary function types we can finally start building the n-nary counterparts of the combinators we discussed at length in Section 2.1.

6.1 Quantifiers

If we already know how to quantify over one variable, we can easily describe how to quantify over n variables by induction over n. This is what quant_n does. Provided a (level polymorphic) quantifier Q and a Set-valued n-ary function f, we distinguish two cases: if n is 0 then the function is already a Set and we can return it directly; otherwise we use Q to quantify over the outer variable which we call x and

proceed to quantify over the remaining variables in $(f\ x)$ by using the induction hypothesis.

```
quantₙ : (∀ {i l} {I : Set i} → (I → Set l) → Set (i ⊔ l)) →
         ∀ n {ls} {as : Sets n ls} →
         Arrows n as (Set r) → Set (r ⊔ (⨆ n ls))
quantₙ Q zero    f = f
quantₙ Q (suc n) f = Q (λ x → quantₙ Q n (f x))
```

We can define the specific instances of n-ary quantification we are interested in by partially applying quant_n with the appropriate concrete quantifiers. Because we are dealing with Set-valued functions, we can leave their arity as an implicit argument and let Agda infer it at use site. In all cases we give them the same name as their unary counterparts as they can be used as drop-in replacements for them.

We start with the n-ary existential quantifier defined using the unary quantifier we introduced in Section 2.1.1.

```
∃⟨_⟩ : Arrows n {ls} as (Set r) → Set (r ⊔ (⨆ n ls))
∃⟨_⟩ = quantₙ Unary.∃⟨_⟩ _
```

Similarly we can define the explicit and implicit universal quantifiers.

```
Π[_] : Arrows n {ls} as (Set r) → Set (r ⊔ (⨆ n ls))
Π[_] = quantₙ Unary.Π[_] _

∀[_] : Arrows n {ls} as (Set r) → Set (r ⊔ (⨆ n ls))
∀[_] = quantₙ Unary.∀[_] _
```

6.2 Pointwise liftings

Pointwise lifting of a binary function can be defined uniformly for any operation of type $(A \to B \to C)$ and any pair of n-ary functions whose domains match and codomains are respectively A and B. It is defined by induction on the arity n of the input functions.

```
lift₂ : ∀ n {ls} {as : Sets n ls} → (A → B → C) →
        Arrows n as A → Arrows n as B → Arrows n as C
lift₂ zero    op f g = op f g
lift₂ (suc n) op f g = λ x → lift₂ n op (f x) (g x)
```

From this very general definition we can recover the combinators we are used to. For each one of them we are able to leave out the arity argument thanks to the observation we made in Section 4.4: Set and $(?A \to ?B)$ are anti-unifiable and Agda is therefore able to reconstruct the arity for us!

Implication is the lifting of the function space.

```
_⇒_ : Arrows n {ls} as (Set r) → Arrows n as (Set s) →
      Arrows n as (Set (r ⊔ s))
_⇒_ = lift₂ _ (λ A B → (A → B))
```

Conjunction is the lifting of pairing.

```
_∩_ : Arrows n {ls} as (Set r) → Arrows n as (Set s) →
      Arrows n as (Set (r ⊔ s))
_∩_ = lift₂ _ (λ A B → (A × B))
```

Disjunction is the lifting of the sum type.

```
_∪_ : Arrows n {ls} as (Set r) → Arrows n as (Set s) →
      Arrows n as (Set (r ⊔ s))
_∪_ = lift₂ _ (λ A B → (A ⊎ B))
```

Negation is obviously not a binary operation. In practice, rather than having multiple ad-hoc lifting functions for various arities we have a fully generic lift_n functional which lifts a k-ary operator to work with k n-ary functions whose respective codomains match the domains of the operator. Its type could be summarised as:

$$
\begin{aligned}
\mathsf{lift}_n : \forall\ k\ n.\\
&(B_1 \to \cdots \to B_k \to B) \to\\
&(A_1 \to \cdots \to A_n \to B_1) \to\\
&\qquad\vdots\\
&(A_1 \to \cdots \to A_n \to B_k) \to\\
&(A_1 \to \cdots \to A_n \to B)
\end{aligned}
$$

The thus generalised definition has a fairly unreadable type so we leave this formal definition out of the paper. Curious readers can consult the accompanying code. We can evidently use lift_n with k equal to 1 to lift negation from an operation on Set to an operation on Arrows.

```
¬_ : Arrows n {ls} as (Set r) → Arrows n as (Set r)
¬_ = liftₙ 1 _ (λ A → (A → ⊥))
```

6.3 Adjustments To The Ambient Indices

We now have obtained the generalised versions of the index-threading combinators we wanted. We can similarly define a number of index-altering combinators. The first two are the n-ary versions of the two operators we described in Section 2.1.3.

Lifting a mere value to a constant n-ary function is a matter of composing const with itself n times.

```
constₙ : ∀ n {ls} {as : Sets n ls} → B → Arrows n as B
constₙ zero    = id
constₙ (suc n) = const ∘ (constₙ n)
```

Updates are a bit more subtle: now that we are not limited to a single index, we can choose which index should be updated. We expect the user to provide a natural number n to target a specific index, the type of the combinator then clearly states that n sets are skipped, the target is updated and the rest of the type is unchanged.

```
_%=_⊢_ : ∀ n {ls} {as : Sets n ls} → (I → J) →
         Arrows n as (J → B) → Arrows n as (I → B)
zero  %= f ⊢ g = g ∘ f
suc n %= f ⊢ g = (n %= f ⊢_) ∘ g
```

The added complexity of working with n-ary relations means that we have more interesting operators than simply the generalised version of the ones we had introduced for unary predicates.

We may for instance want to map a unary function on the result of an n-ary one. Note that this empowers us to partially apply any n-ary function to a value x in its k-th argument by choosing to see it as a k-ary function and mapping $(\_\ \$\ x)$ on it.

```
map_n : ∀ n {ls} {as : Sets n ls} →
        (B → C) → Arrows n as B → Arrows n as C
map_n zero    f v = f v
map_n (suc n) f g = map_n n f ∘ g
```

7 Congruence and Substitution

So far the types we have ascribed our combinators for n-ary relations where fairly tame. Things get a bit more complicated when dealing with congruence and substitution: we won't be able to write these functions' types directly. Both definitions follow the same structure: we start by computing the operation's type by induction and we can then implement the operation itself.

7.1 Congruence

The type of congruence mentions only one function. However it is applied to two distinct lists of values to form the left-hand side and the right-hand side of the conclusion. As a consequence when we compute the type we take two functions as inputs and use one to apply to the arguments meant for the left-hand side and the other for the ones meant for the right-hand side of the equation.

Congruence for two 0-ary functions collapses to simply propositional equality of the two constant values.

Congruence for two (suc n)-ary functions f and g amounts to stating that for any pair of equal values x and y we expect that $(f\ x)$ and $(g\ y)$ are congruent.

```
Cong_n : ∀ n {ls} {as : Sets n ls} {R : Set r} →
         (f g : Arrows n as R) → Set (r ⊔ (⨆ n ls))
Cong_n zero    f g = f ≡ g
Cong_n (suc n) f g = ∀ {x y} → x ≡ y → Cong_n n (f x) (g y)
```

The congruence lemma is then obtained by stating that the n-ary function f is congruent with itself. We prove it by induction on n, pattern-matching on the proofs of equality as we go along.

```
cong_n : ∀ n {ls} {as : Sets n ls} {R : Set r} →
         (f : Arrows n as R) → Cong_n n f f
cong_n zero    f    = refl
cong_n (suc n) f refl = cong_n n (f _)
```

7.2 Substitution

The definition of Subst_n is identical to that of Cong_n except for the base case: instead of f and g being two values, they are two Sets. We demand a function transporting values in f to ones in g.

```
Subst_n : ∀ n {ls} {as : Sets n ls} →
          (f g : Arrows n as (Set r)) → Set (r ⊔ (⨆ n ls))
Subst_n zero    f g = f → g
Subst_n (suc n) f g = ∀ {x y} → x ≡ y → Subst_n n (f x) (g y)
```

Substitution acts on n-ary relations. Recalling our observation made in Section 4.4 that Agda can easily reconstruct the arity of Set-valued functions, we can make n an implicit argument.

```
subst_n : ∀ {n r ls} {as : Sets n ls} →
          (f : Arrows n as (Set r)) → Subst_n n f f
subst_n {zero}  f x    = x
subst_n {suc n} f refl = subst_n (f _)
```

8 Further Generic Programming Efforts

The small language we have developed to talk about n-ary functions can be used beyond our first few motivating examples of congruence, substitution, and combinators to define types involving relations. We detail in this section various results which fall out naturally from this work. We start with generic currying and uncurrying which we can then use to define an n-ary zipWith and revisit printf in direct style.

8.1 Product and (Un)Currying

We gave in Section 5 a semantics to our reified types as proper n-ary function types. We can alternatively interpret a Sets as a big right-nested and ⊤-terminated product containing one value for each Set. We once more proceed by induction on n.

```
Product : ∀ n {ls} → Sets n ls → Set (⨆ n ls)
Product zero    _        = ⊤
Product (suc n) (a , as) = a × Product n as
```

We can convert back and forth between a unary function whose domain is a Product of Sets and an n-ary function whose domains are the same sets. These conversion functions correspond to currying and uncurrying. Both curry_n and $\mathsf{uncurry}_n$ are implemented by structural induction on n and in terms of their binary counterparts. In the base case, the function is either applied to tt or uses const to throw away a value of type ⊤; this is an artefact of the fact our definition of Product is ⊤-terminated

```
curry_n : ∀ n {ls} {as : Sets n ls} →
          (Product n as → R) → Arrows n as R
curry_n zero    f = f _
curry_n (suc n) f = curry_n n ∘ curry f

uncurry_n : ∀ n {ls} {as : Sets n ls} →
            Arrows n as R → (Product n as → R)
uncurry_n zero    f = const f
uncurry_n (suc n) f = uncurry (uncurry_n n ∘ f)
```

⊤-free Variant In practice users do not tend to write ⊤-terminated right-nested products. As a consequence it is convenient to have a definition of Product which has a special case for Sets of size exactly 1 returning the Set without pairing it with ⊤. This makes curry_n and $\mathsf{uncurry}_n$ more useful overall. Most generic functions however are easier to implement using the ⊤-terminated version of Product. In our library we provide both as well as conversion functions between the two interpretations.

8.2 N-ary Zipping Functions

Some functions are easier to write curried but nicer to use uncurried. This is the case with $\mathsf{zipWith}_n$, the n-ary version of the higher-order function which takes a function and two lists as inputs and produces a list by processing both lists in lockstep and using the function it was passed to combine their elements. Using ellipses, we would write its type as:

$$\mathsf{zipWith}_n : \forall\ n.\ (A_1 \to \cdots \to A_n \to B) \to \\ \mathsf{List}\ A_1 \to \cdots \to \mathsf{List}\ A_n \to \mathsf{List}\ B$$

To formally write this type, we need to explain how to map a level polymorphic endofunctor on Set (here: List) over a (Sets n ls). We proceed by induction on n.

```
_<$>_ : (∀ {l} → Set l → Set l) →
        ∀ {n ls} → Sets n ls → Sets n ls
_<$>_ f {zero}  as       = _
_<$>_ f {suc n} (a , as) = f a , f <$> as
```

As explained earlier it is vastly easier to implement the function using the uncurried type, and to then recover the desired type by invoking generic (un)currying in the appropriate places. The function we want is therefore implemented in term of an auxiliary definition called zw-aux.

```
zipWith_n : ∀ n {ls} {as : Sets n ls} →
            Arrows n as R → Arrows n (List <$> as) (List R)
zipWith_n n f = curry_n n (zw-aux n (uncurry_n n f))
```

Implementation The auxiliary definition is still a bit involved so we detail each equation of its definition below. We start with its type first.

```
zw-aux : ∀ n {ls} {as : Sets n ls} →
         (Product n as → R) →
         (Product n (List <$> as) → List R)
```

When n is 0, a Haskeller would typically return an infinite list containing the value f repeated. However this is not possible in Agda, a total language [13]: all the lists have to be finite. Our only principled option is to return the empty one.

```
zw-aux 0 f as = []
```

Because the behaviour of the 0 case is less than ideal, we bypass it every time except if $\mathsf{zipWith}_n$ is explicitly called on 0. This is done by having a special case for n equals 1. In this situation, we can get our hands on a function f of type $((A \times \top) \to R)$ and a List A and we need to return a List R. We map a tweaked version of the function on the list.

```
zw-aux 1 f (as , _) = map (f ∘ (_ , tt)) as
```

The meat of the definition is in the last case: we are given a function $((A \times A_0 \times \cdots \times A_n) \to R)$, a list of As and a product of lists $(\mathsf{List}\ A_0 \times \cdots \times \mathsf{List}\ A_n)$. We massage the function to obtain another one of type $((A_0 \times \cdots \times A_n) \to (A \to R))$ which we can combine with the product of lists thanks to our induction hypothesis. This gives us back a list of functions of type $(A \to R)$. We can conclude thanks to the usual binary zipWith to combine this list of functions with the list of arguments we already had.

```
zw-aux (suc n) f (as , ass) = zipWith _$_ fs as
  where fs = zw-aux n (flip (curry f)) ass
```

8.3 Printf

The definitions we have introduced also make it easy to implement Printf in direct style as opposed to the classic accumulator-based definition [1, 4]. We work in a simplified setting which allows us to focus on the contribution our n-ary combinators bring to the table. Our printf will only take natural numbers as arguments and we will not worry about defining the lexer transforming a raw string into a Format, that is to say a list of Chunks each being either a Nat corresponding to a "%u" directive (i.e. unsigned decimal integer) or a Raw string.

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

Format : Set
Format = List Chunk
```

Our notion of Format is not intrinsically sized but we do need to know how many arguments our printf function is going to take if we want to use the machinery for n-ary functions. We assume the existence of a size function counting the number of Nat in a Format. We also assume the existence of 0ℓs, a (Levels n) equal to 0ℓ everywhere. Using these we can give Format a semantics as a Sets of arguments printf will expect. To each Nat we associate a $\mathbb{N}$ constraint, the other Chunks do not give rise to the need for an input.

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

The essence of printf is then given by a function assemble which collects a list of strings from various sources. Whenever the format expects a natural number, we know we got one as an input and can show it. Otherwise the raw string to use is specified in the Format itself as an argument to Raw.

```
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
```

The toplevel function is obtained by currying the composition of concat and assemble.

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

We can check on an example that we do get a function with the appropriate type when we use a concrete Format (here the one we would obtain from the string "%u < %u").

```
lessThan : ℕ → ℕ → String
lessThan = printf (Nat :: Raw " < " :: Nat :: [])
```

And that it does produce the expected string when run on arguments.

```
_ : lessThan 2 5 ≡ "2 < 5"
_ = refl
```

9 Conclusion, Related and Future Work

We have seen that Agda's standard library defines a useful couple of functions to produce proofs of equality as well as a type-level domain specific language to manipulate unary predicates. We then got acquainted with the unifier and the process by which a unification constraint can lead to the reconstruction of a function's implicit arguments. Based on this knowledge we have designed a representation of n-ary function types particularly amenable to such reconstructions. This allowed us to define n-ary versions of congruence, substitution as well as the basic building blocks of the type-level DSL for relations we longed for. The notions introduced to set the stage for these definitions were already powerful enough to allow us to revisit classic dependently typed traversals such as an n-ary version of zipWith, or direct-style printf.

Limitations We are relying heavily on two key features of Agda which are not implemented in other dependently typed as far as we know. First of all Levels are a first class notion, they can be manipulated, stored in data structures, passed around and computed with just like any other primitive type. Second, Agda's unifier has a (principled!) heuristics which attempts to invert stuck functions when solving constraints. Without the first feature we would be unable to write our representation and without the second the user would be burdened with always having to mention the arity of the relation they are manipulating. We hope that our detailed use-case incites other languages to implement similar features, and users to find new ways to exploit them.

Codes for N-ary Function Types We can find in the literature various deep [14] and shallow [10, 15] embeddings of polymorphic types and a fortiori of n-ary function types in a dependently typed language. However none of them are fully level polymorphic and most only consider the representation as a secondary requirement, their focus being on certifying equivalent programs in Generic Haskell [7]. We however care deeply about level polymorphism as well as being unification-friendly to minimise the reification work the user needs to do.

Telescopes The lack of dependencies between the various domains and the codomain of our Arrows is flagrant. A natural question to ask is how much of this machinery can be generalised to telescopes rather than mere Sets without incurring any additional burden on the user. From experience we know that it is sometimes wise to explicitly use the non-dependent version of an operator (e.g. function composition) to inform Agda's unifier that it is only looking for a solution in a restricted subset.

Datatype genericity Our implementation of an n-ary version of zipWith started as drive-by generic programming, demonstrating that the notions introduced for our purposes could be useful in a more general context. This result is not new either with or without dependent types [6, 10]. It can however be extended as previous efforts in dependently typed programming have demonstrated: Weirich and Casinghino's work [15] on arity-generic but also data-generic programming suggest we should be able to push this further. Their development predates the addition of universe polymorphism to Agda and although the traversals are adequately heterogeneous, their approach would not scale to universe polymorphic functions.

References

[1] Lennart Augustsson. 1998. Cayenne - a Language with Dependent Types. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 239–250. https://doi.org/10.1145/289423.289451

[2] Jesper Cockx. 2017. *Dependent Pattern Matching and Proof-Relevant Unification.* Ph.D. Dissertation. Katholieke Universiteit Leuven, Belgium. https://lirias.kuleuven.be/handle/123456789/583556

[3] Jesper Cockx and Dominique Devriese. 2018. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *J. Funct. Program.* 28 (2018), e12. https://doi.org/10.1017/S095679681800014X

[4] Olivier Danvy. 1998. Functional Unparsing. *J. Funct. Program.* 8, 6 (1998), 621–625. http://journals.cambridge.org/action/displayAbstract?aid=44189

[5] Peter Dybjer. 1994. Inductive Families. *Formal Asp. Comput.* 6, 4 (1994), 440–465. https://doi.org/10.1007/BF01211308

[6] Daniel Fridlender and Mia Indrika. 2000. Do we need dependent types? *J. Funct. Program.* 10, 4 (2000), 409–415. http://journals.cambridge.org/action/displayAbstract?aid=59741

[7] Ralf Hinze. 2000. A New Approach to Generic Functional Programming. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, Mark N. Wegman and Thomas W. Reps (Eds.). ACM, 119–132. https://doi.org/10.1145/325694.325709

[8] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (1996), 196. https://doi.org/10.1145/242224.242477

[9] Per Martin-Löf. 1982. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics* 104 (1982), 153–175.

[10] Conor McBride. 2002. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.* 12, 4&5 (2002), 375–392. https://doi.org/10.1017/S0956796802004355

[11] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

[12] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures (Lecture Notes in Computer Science)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.), Vol. 5832. Springer, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

[13] D. A. Turner. 2004. Total Functional Programming. *J. UCS* 10, 7 (2004), 751–768. https://doi.org/10.3217/jucs-010-07-0751

[14] Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. 2008. Polytypic programming in COQ. In *Proceedings of the ACM SIGPLAN Workshop on Genetic Programming, WGP 2008, Victoria, BC, Canada, September 20, 2008*, Ralf Hinze and Don Syme (Eds.). ACM, 49–60. https://doi.org/10.1145/1411318.1411326

[15] Stephanie Weirich and Chris Casinghino. 2010. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM Workshop Programming Languages meets Program Verification, PLPV 2010, Madrid, Spain, January 19, 2010*, Jean-Christophe Filliâtre and Cormac Flanagan (Eds.). ACM, 15–26. https://doi.org/10.1145/1707790.1707799