The Gentle Art of Levitation

James Chapman
Institute of Cybernetics, Tallinn
University of Technology
james@cs.ioc.ee

Pierre-Évariste Dagand
Conor McBride
University of Strathclyde
{dagand,conor}@cis.strath.ac.uk

Peter Morris
University of Nottingham
pwm@cs.nott.ac.uk

Abstract

We present a closed dependent type theory whose inductive types are given not by a scheme for generative declarations, but by encoding in a *universe*. Each inductive datatype arises by interpreting its *description*—a first-class value in a datatype of descriptions. Moreover, the latter itself has a description. Datatype-generic programming thus becomes ordinary programming. We show some of the resulting generic operations and deploy them in particular, useful ways on the datatype of datatype descriptions itself. Simulations in existing systems suggest that this apparently self-supporting setup is achievable without paradox or infinite regress.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Language Constructs and Features*]: Data types and structures

General Terms Design, Languages, Theory

1. Introduction

Dependent datatypes, such as the ubiquitous vectors (lists indexed by length) express *relative* notions of data validity. They allow us to function in a complex world with a higher standard of basic hygiene than is practical with the context-free datatypes of ML-like languages. Dependent type systems, as found in Agda [Norell 2007], Coq [The Coq Development Team], Epigram [McBride and McKinna 2004], and contemporary Haskell [Cheney and Hinze 2003; Xi et al. 2003], are beginning to make themselves useful. As with rope, the engineering benefits of type indexing sometimes outweigh the difficulties you can arrange with enough of it.

The blessing of expressing just the right type for the job can also be a curse. Where once we might have had a small collection of basic datatypes and a large library, we now must cope with a cornucopia of finely confected structures, subtly designed, subtly different. The basic vector equipment is much like that for lists, but we implement it separately, often retyping the same code. The Agda standard library [Danielsson 2010], for example, sports a writhing mass of list-like structures, including vectors, bounded-length lists, difference lists, reflexive-transitive closures—the list is petrifying. Here, we seek equipment to tame this gorgon's head with *reflection*.

The business of belonging to a datatype is itself a notion relative to the type's *declaration*. Most typed functional languages, including those with dependent types, feature a datatype declaration construct, external to and extending the language for defining values and programs. However, dependent type systems also allow us to reflect types as the image of a function from a set of 'codes'—a *universe construction* [Martin-Löf 1984]. Computing with codes, we expose operations on and relationships between the types they reflect. Here, we adopt the universe as our guiding design principle. We abolish the datatype declaration construct, by reflecting it as a datatype of datatype descriptions which, moreover, *describes itself*. This apparently self-supporting construction is a trick, of course, but we shall show the art of it. We contribute

- a *closed* type theory, extensible only *definitionally*, nonetheless equipped with a universe of inductive families of datatypes;
- a *self-encoding* of the universe codes as a datatype in the universe—datatype generic programming is just programming;
- a bidirectional *type propagation* mechanism to conceal artefacts of the encoding, restoring a convenient presentation of data;
- examples of generic operations and constructions over our universe, notably the *free monad* construction;
- datatype generic programming delivered *directly*, not via some isomorphic model or 'view' of declared types.

We study two universes as a means to explore this novel way to equip a programming language with its datatypes. We warm up with a universe of *simple* datatypes, just sufficient to describe itself. Once we have learned this art, we scale up to *indexed* datatypes, encompassing the inductive families [Dybjer 1991; Luo 1994] found in Coq and Epigram, and delivering experiments in generic programming with applications to the datatype of codes itself.

We aim to deliver proof of concept, showing that a closed theory with a self-encoding universe of datatypes can be made practicable, but we are sure there are bigger and better universes waiting for a similar treatment. Benke, Dybjer and Jansson [Benke et al. 2003] provide a useful survey of the possibilities, including extension to inductive-recursive definition, whose closed-form presentation [Dybjer and Setzer 1999, 2000] is both an inspiration for the present enterprise, and a direction for future study.

The work of Morris, Altenkirch and Ghani [Morris 2007; Morris and Altenkirch 2009; Morris et al. 2009] on (indexed) containers has informed our style of encoding and the equipment we choose to develop, but the details here reflect pragmatic concerns about intensional properties which demand care in practice. We have thus been able to implement our work as the basis for datatypes in the Epigram 2 prototype [Brady et al.]. We have also developed a *stratified* model of our coding scheme in Agda and Coq[1].

[1] This model is available at
`http://personal.cis.strath.ac.uk/~dagand/levitate.tar.gz`

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

2. The Type Theory

One challenge in writing this paper is to extricate our account of datatypes from what else is new in Epigram 2. In fact, we demand relatively little from the setup, so we shall start with a 'vanilla' theory and add just what we need. The reader accustomed to dependent types will recognise the basis of her favourite system; for those less familiar, we try to keep the presentation self-contained.

2.1 Base theory

We adopt a traditional presentation for our type theory, with three mutually defined systems of judgments: *context validity*, *typing*, and *equality*, with the following forms:

$\Gamma \vdash \textsc{valid}$	Γ is a valid context, giving types to variables
$\Gamma \vdash t : T$	term t has type T in context Γ
$\Gamma \vdash s \equiv t : T$	s and t are equal at type T in context Γ

The rules are formulated to ensure that the following 'sanity checks' hold by induction on derivations

$$\begin{array}{l} \Gamma \vdash t : T \quad \Rightarrow \Gamma \vdash \textsc{valid} \wedge \Gamma \vdash T : \textsc{Set} \\ \Gamma \vdash s \equiv t : T \Rightarrow \Gamma \vdash s : T \wedge \Gamma \vdash t : T \end{array}$$

and that judgments J are preserved by well-typed instantiation.

$$\Gamma; x : S; \Delta \vdash J \Rightarrow \Gamma \vdash s : S \Rightarrow \Gamma; \Delta[s/x] \vdash J[s/x]$$

We specify equality as a judgment, leaving open the details of its implementation, requiring only a congruence including ordinary computation (β-rules), decided, e.g., by testing α-equivalence of β-normal forms [Adams 2006]. Coquand and Abel feature prominently in a literature of richer equalities, involving η-expansion, proof-irrelevance and other attractions [Abel et al.; Coquand 1996]. Agda and Epigram 2 support such features, Coq currently does not, but they are surplus to requirements here.

Context validity ensures that variables inhabit well-formed sets.

$$\frac{}{\vdash \textsc{valid}} \qquad \frac{\Gamma \vdash S : \textsc{Set}}{\Gamma; x : S \vdash \textsc{valid}}\ x \notin \Gamma$$

The basic typing rules for tuples and functions are also standard, save that we locally adopt $\textsc{Set} : \textsc{Set}$ for presentational purposes. Usual techniques to resolve this *typical ambiguity* apply [Courant 2002; Harper and Pollack; Luo 1994]. A formal treatment of stratification for our system is a matter of ongoing work.

$$\frac{\Gamma; x : S; \Delta \vdash \textsc{valid}}{\Gamma; x : S; \Delta \vdash x : S} \qquad \frac{\Gamma \vdash s : S \quad \Gamma \vdash S \equiv T : \textsc{Set}}{\Gamma \vdash s : T}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \textsc{Set} : \textsc{Set}} \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{1} : \textsc{Set}} \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash [] : \mathsf{1}}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma; x : S \vdash T : \textsc{Set}}{\Gamma \vdash (x : S) \times T : \textsc{Set}}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma; x : S \vdash T : \textsc{Set} \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash [s, t]_{x.T} : (x : S) \times T}$$

$$\frac{\Gamma \vdash p : (x : S) \times T}{\Gamma \vdash \pi_0\, p : S} \qquad \frac{\Gamma \vdash p : (x : S) \times T}{\Gamma \vdash \pi_1\, p : T[\pi_0\, p/x]}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma; x : S \vdash T : \textsc{Set}}{\Gamma \vdash (x : S) \to T : \textsc{Set}}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma; x : S \vdash t : T}{\Gamma \vdash \lambda_S x.t : (x : S) \to T} \qquad \frac{\Gamma \vdash f : (x : S) \to T \quad \Gamma \vdash s : S}{\Gamma \vdash f\, s : T[s/x]}$$

Notation. We subscript information needed for type synthesis but not type checking, e.g., the domain of a λ-abstraction, and suppress it informally where clear. Square brackets denote tuples, with a LISP-like right-nesting convention: $[a\ b]$ abbreviates $[a, [b, []]]$.

The judgmental equality comprises the computational rules below, closed under reflexivity, symmetry, transitivity and structural congruence, even under binders. We omit the mundane rules which ensure these closure properties for reasons of space.

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma; x : S \vdash t : T \quad \Gamma \vdash s : S}{\Gamma \vdash (\lambda_S x.t)\, s \equiv t[s/x] : T[s/x]}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma; x : S \vdash T : \textsc{Set} \quad \Gamma; s : S \vdash t : T[s/x]}{\Gamma \vdash \pi_0\, ([s, t]_{x.T}) \equiv s : S} \qquad \frac{\Gamma \vdash s : S \quad \Gamma; x : S \vdash T : \textsc{Set} \quad \Gamma; s : S \vdash t : T[s/x]}{\Gamma \vdash \pi_1\, ([s, t]_{x.T}) \equiv t : T[s/x]}$$

Given a suitable stratification of $\textsc{Set}$, the computation rules yield a terminating evaluation procedure, ensuring the decidability of equality and thence type checking.

2.2 Finite enumerations of tags

It is time for our first example of a *universe*. You might want to offer a choice of named constructors in your datatypes: we shall equip you with sets of tags to choose from. Our plan is to implement (by extending the theory, or by encoding) the signature

$$\mathsf{En} : \textsc{Set} \qquad \#(E : \mathsf{En}) : \textsc{Set}$$

where some value $E : \mathsf{En}$ in the 'enumeration universe' describes a type of tag choices $\#E$. We shall need some tags—valid identifiers, marked to indicate that they are data, not variables scoped and substitutable—so we hardwire these rules:

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{Tag} : \textsc{Set}} \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \text{'}s : \mathsf{Tag}}\ s \text{ a valid identifier}$$

Let us describe enumerations as lists of tags, with signature:

$$\mathsf{nE} : \mathsf{En} \qquad \mathsf{cE}\,(t : \mathsf{Tag})\,(E : \mathsf{En}) : \mathsf{En}$$

What are the *values* in $\#E$? Formally, we represent the choice of a tag as a numerical index into E, via new rules:

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash 0 : \#(\mathsf{cE}\, t\, E)} \qquad \frac{\Gamma \vdash n : \#E}{\Gamma \vdash 1{+}n : \#(\mathsf{cE}\, t\, E)}$$

However, we expect that in practice, you might rather refer to these values *by tag*, and we shall ensure that this is possible in due course.

Enumerations come with further machinery. Each $\#E$ needs an eliminator, allowing us to branch according to a tag choice. Formally, whenever we need such new computational facilities, we add primitive operators to the type theory and extend the judgmental equality with their computational behavior. However, for compactness and readability, we shall write these operators as functional programs (much as we model them in Agda).

We first define the 'small product' π operator:

$$\begin{array}{l} \pi\ : (E : \mathsf{En})(P : \#E \to \textsc{Set}) \to \textsc{Set} \\ \pi\, \mathsf{nE} \qquad P \mapsto \mathsf{1} \\ \pi\, (\mathsf{cE}\, t\, E)\ P \mapsto P\, 0 \times \pi\, E\, \lambda x.P\,(1{+}x) \end{array}$$

This builds a right-nested tuple type, packing a $P\,i$ value for each i in the given domain. The step case exposes our notational convention that binders scope rightwards as far as possible. These tuples are 'jump tables', tabulating dependently typed functions. We give this functional interpretation—the eliminator we need—by the switch operator, which, unsurprisingly, iterates projection:

$$\begin{array}{l} \mathsf{switch} : (E : \mathsf{En})(P : \#E \to \textsc{Set}) \to \pi\, E\, P \to (x : \#E) \to P\, x \\ \mathsf{switch}\, (\mathsf{cE}\, t\, E)\, P\, b\, 0 \qquad \mapsto \pi_0\, b \\ \mathsf{switch}\, (\mathsf{cE}\, t\, E)\, P\, b\, (1{+}x) \mapsto \mathsf{switch}\, E\, (\lambda x.P(1{+}x))\, (\pi_1\, b)\, x \end{array}$$

The π and switch operators deliver dependent elimination for finite enumerations, but are rather awkward to use directly. We do

$\boxed{\Gamma \Vdash \mathsf{exprEx} \triangleright \mathsf{term} \in \mathsf{type}}$

$$\frac{\Gamma \Vdash \textsc{Set} \ni T \triangleright T' \quad \Gamma \Vdash T' \ni t \triangleright t'}{\Gamma \Vdash (t : T) \triangleright t' \in T'}$$

$$\frac{\Gamma; x : S; \Delta \vdash \textsc{valid}}{\Gamma; x : S; \Delta \Vdash x \triangleright x \in S} \qquad \frac{\Gamma \Vdash f \triangleright f' \in (x : S) \to T \quad \Gamma \Vdash S \ni s \triangleright s'}{\Gamma \Vdash f\, s \triangleright f'\, s' \in T[s'/x]}$$

$$\frac{\Gamma \Vdash p \triangleright p' \in (x : S) \times T}{\Gamma \Vdash \pi_0\, p \triangleright \pi_0\, p' \in S} \qquad \frac{\Gamma \Vdash p \triangleright p' \in (x : S) \times T}{\Gamma \Vdash \pi_1\, p \triangleright \pi_1\, p' \in T[\pi_0\, p'/x]}$$

Figure 1. Type synthesis

not write the range for a λ-abstraction, so it is galling to supply P for functions defined by switch. Let us therefore find a way to recover the tedious details of the encoding from types.

2.3 Type propagation

Our approach to tidying the coding cruft is deeply rooted in the bidirectional presentation of type checking from Pierce and Turner [Pierce and Turner 1998]. They divide type inference into two communicating components. In *type synthesis*, types are *pulled* out of terms. A typical example is a variable in the context:

$$\frac{\Gamma; x : S; \Delta \vdash \textsc{valid}}{\Gamma; x : S; \Delta \vdash x : S}$$

Because the context stores the type of the variable, we can extract the type whenever the variable is used.

On the other hand, in the *type checking* phase, types are *pushed* into terms. We are handed a type together with a term, our task consists of checking that the type admits the term. In doing so, we can and should use the information provided by the type. Therefore, we can relax our requirements on the term. Consider λ-abstraction:

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma; x : S \vdash t : T}{\Gamma \vdash \lambda_S x.\, t : (x : S) \to T}$$

The official rules require an annotation specifying the domain. However, in type *checking*, the Π-type we push in determines the domain, so we can drop the annotation.

We adapt this idea, yielding a *type propagation* system, whose purpose is to elaborate compact *expressions* into the terms of our underlying type theory, much as in the definition of Epigram 1 [McBride and McKinna 2004]. We divide expressions into two syntactic categories: exprIn into which types are pushed, and exprEx from which types are extracted. In the bidirectional spirit, the exprIn are subject to type *checking*, while the exprEx—variables and elimination forms—admit type *synthesis*. We embed exprEx into exprIn, demanding that the synthesised type coincides with the type proposed. The other direction—only necessary to apply abstractions or project from pairs—takes a type annotation.

Type synthesis (Fig. 1) is the *source* of types. It follows the exprEx syntax, delivering both the elaborated term and its type. Terms and expressions never mix: e.g., for application, we instantiate the range with the *term* delivered by checking the argument *expression*. Hardwired operators are checked as variables.

Dually, type checking judgments (Fig. 2) are *sinks* for types. From an exprIn and a type pushed into it, they elaborate a low-level term, extracting information from the type. Note that we inductively ensure the following 'sanity checks':

$$\Gamma \Vdash e \triangleright t \in T \Rightarrow \Gamma \vdash t : T$$
$$\Gamma \Vdash T \ni e \triangleright t \Rightarrow \Gamma \vdash t : T$$

$\boxed{\Gamma \Vdash \mathsf{type} \ni \mathsf{exprIn} \triangleright \mathsf{term}}$

$$\frac{\Gamma \Vdash s \triangleright s' \in S \quad \Gamma \Vdash \textsc{Set} \ni S \equiv T}{\Gamma \Vdash T \ni s \triangleright s'}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \Vdash \textsc{Set} \ni \textsc{Set} \triangleright \textsc{Set}}$$

$$\frac{\Gamma \Vdash \textsc{Set} \ni S \triangleright S' \quad \Gamma; x : S' \Vdash \textsc{Set} \ni T \triangleright T'}{\Gamma \Vdash \textsc{Set} \ni (x : S) \to T \triangleright (x : S') \to T'}$$

$$\frac{\Gamma; x : S \Vdash T \ni t \triangleright t'}{\Gamma \Vdash (x : S) \to T \ni \lambda x.\, t \triangleright \lambda_S x.\, t'}$$

$$\frac{\Gamma \Vdash \textsc{Set} \ni S \triangleright S' \quad \Gamma; x : S' \Vdash \textsc{Set} \ni T \triangleright T'}{\Gamma \Vdash \textsc{Set} \ni (x : S) \times T \triangleright (x : S') \times T'}$$

$$\frac{\Gamma \Vdash S \ni s \triangleright s' \quad \Gamma \Vdash T[s'/x] \ni t \triangleright t'}{\Gamma \Vdash (x : S) \times T \ni [s, t] \triangleright [s', t']_{x.T}}$$

$$\frac{\Gamma \Vdash (x : S) \to (y : T) \to U[[x, y]_{x.T}/p] \ni f \triangleright f'}{\Gamma \Vdash (p : (x : S) \times T) \to U \ni \wedge f \triangleright \lambda_{((x S) \times T)} p.\, f'\, (\pi_0\, p)\, (\pi_1\, p)}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \Vdash \textsc{Set} \ni 1 \triangleright 1} \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \Vdash 1 \ni [] \triangleright []}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \Vdash \mathsf{En} \ni [] \triangleright \mathsf{nE}} \qquad \frac{\Gamma \Vdash \mathsf{En} \ni E \triangleright E'}{\Gamma \Vdash \mathsf{En} \ni [\text{'}t, E] \triangleright \mathsf{cE}\ \text{'}t\ E'}$$

$$\frac{\Gamma \vdash E : \mathsf{En}}{\Gamma \Vdash \#(\mathsf{cE}\ \text{'}t\ E) \ni \text{'}t \triangleright 0} \qquad \frac{\Gamma \Vdash \#E \ni \text{'}t \triangleright n \quad \text{'}t \neq \text{'}t_0}{\Gamma \Vdash \#(\mathsf{cE}\ \text{'}t_0\ E) \ni \text{'}t \triangleright 1{+}n}$$

$$\frac{\Gamma \vdash E : \mathsf{En}}{\Gamma \Vdash \#(\mathsf{cE}\ \text{'}t\ E) \ni 0 \triangleright 0} \qquad \frac{\Gamma \Vdash \#E \ni n \triangleright n'}{\Gamma \Vdash \#(\mathsf{cE}\ \text{'}t_0\ E) \ni 1{+}n \triangleright 1{+}n'}$$

$$\frac{\Gamma \Vdash \pi\, E\, (\lambda_{\#E} x.\, T) \ni \overrightarrow{[t]} \triangleright t'}{\Gamma \Vdash (x : \#E) \to T \ni \overrightarrow{[t]} \triangleright \mathsf{switch}\, E\, (\lambda_{\#E} x.\, T)\, t'}$$

Figure 2. Type checking

Canonical set-formers are *checked*: we could exploit $\textsc{Set} : \textsc{Set}$ to give them synthesis rules, but this would prejudice our future stratification plans. Note that abstraction and pairing are free of annotation, as promised. Most of the propagation rules are unremarkably structural: we have omitted some mundane rules which just follow the pattern, e.g., for Tag.

However, we also add abbreviations. We write $\wedge f$, pronounced 'uncurry f' for the function which takes a pair and feeds it to f one component at a time, letting us name them individually. Now, for the finite enumerations, we go to work.

Firstly, we present the codes for enumerations as right-nested tuples which, by our LISP convention, we write as unpunctuated lists of tags $[\text{'}t_0 \ldots \text{'}t_n]$. Secondly, we can denote an element *by its name*: the type pushed in allows us to recover the numerical index. We retain the numerical forms to facilitate *generic* operations and ensure that shadowing is punished fittingly, not fatally. Finally, we express functions from enumerations as tuples. Any tuple-form, [] or $[\_,\_]$, is accepted by the function space—the generalised product—if it is accepted by the small product. Propagation fills in the appeal to switch, copying the range information.

Our interactive development tools also perform the reverse transformation for intelligible output. The encoding of any specific enumeration is thus hidden by these translations. Only, and rightly, in enumeration-generic programs is the encoding exposed.

Our type propagation mechanism does no constraint solving, just copying, so it is just the thin end of the elaboration wedge. It can afford us this 'assembly language' level of civilisation as

En universe specifies not only the *representation* of the low-level values in each set as bounded numbers, but also the *presentation* of these values as high-level tags. To encode only the former, we should merely need the *size* of enumerations, but we extract more work from these types by making them more informative. We have also, *en passant*, distinguished enumerations which have the same cardinality but describe distinct notions: #['red 'blue] is not #['green 'orange].

3. A Universe of Inductive Datatypes

In this section, we describe an implementation of inductive types, as we know them from ML-like languages. By working with familiar datatypes, we hope to focus on the delivery mechanism, warming up gently to the indexed datatypes we really want. Dybjer and Setzer's closed formulation of induction-recursion [Dybjer and Setzer 1999], but without the '-recursion'. An impredicative Church-style encoding of datatypes is not adequate for dependently typed programming, as although such encodings present data as non-dependent eliminators, they do not support dependent *induction* [Geuvers 2001]. Whilst the λ-calculus captures all that data can *do*, it cannot ultimately delimit all that data can *be*.

3.1 The power of Σ

In dependently typed languages, Σ-types can be interpreted as two different generalisations. This duality is reflected in the notation we can find in the literature. The notation $\Sigma_{x:A}(B\,x)$ stresses that Σ-types are 'dependent sums', generalising sums over arbitrary arities, where simply typed languages have finite sums.

On the other hand, our choice, $(x:A)\times(B\,x)$, emphasises that Σ-types generalise products, with the type of the second component depending on the value of the first. Simply typed languages do not express such relative validity.

In ML-like languages, datatypes are presented as a *sum-of-products*. A datatype is defined by a finite sum of constructors, each carrying a product of arguments. To embrace these datatypes, we have to capture this grammar. With dependent types, the notion of sum-of-products translates into *sigmas-of-sigmas*.

3.2 The universe of descriptions

While sigmas-of-sigmas can give a *semantics* for the sum-of-products structure in each node of the tree-like values in a datatype, we need to account for the recursive structure which ties these nodes together. We do this by constructing a *universe* [Martin-Löf 1984]. Universes are ubiquitous in dependently typed programming [Benke et al. 2003; Oury and Swierstra 2008], but here we take them as the foundation of our notion of datatypes.

To add inductive types to our type theory, we build a universe of datatype *descriptions* by implementing the signature presented in Figure 3, with codes mimicking the grammar of datatype declarations. We can read a description $D : \mathsf{Desc}^n$ as a 'pattern functor' on $\textsc{Set}^n$, with $[\![D]\!]$ its action on an object, X, soon to be instantiated recursively. The superscripts indicate the SET-levels at which we *expect* these objects in a stratified system. This is but an informal notation, to give a flavour of the stratified presentation. Note that the functors so described are *strictly positive*, by construction.

Descriptions are sequential structures ending in '1, indicating the empty tuple. To build sigmas-of-sigmas, we provide a 'Σ code, interpreted as a Σ-type. To request a recursive component, we have 'ind× D, where D describes the rest of the node. These codes give us sigmas-of-sigmas with recursive places. An equivalent, more algebraic presentation could be given, as illustrated in Section 5.

We admit to being a little coy, writing of 'implementing a signature' without clarifying how. A viable approach would simply be to extend the theory with constants for the constructors and an operator for $[\![D]\!]$. In Section 4, you will see what we do instead. Meanwhile, let us gain some intuition by developing examples.

$$
\begin{array}{ll}
\mathsf{Desc}^n & : \textsc{Set}^{n+1} \\
\text{'}1 & : \mathsf{Desc}^n \\
\text{'}\Sigma\,(S : \textsc{Set}^n)\,(D : S \to \mathsf{Desc}^n) & : \mathsf{Desc}^n \\
\text{'}\mathsf{ind}\times\,(D : \mathsf{Desc}^n) & : \mathsf{Desc}^n
\end{array}
$$

$$
\begin{array}{ll}
[\![\_]\!] : \mathsf{Desc}^n \to \textsc{Set}^n \to \textsc{Set}^n & \\
[\![\text{'}1]\!] & X \mapsto 1 \\
[\![\text{'}\Sigma\,S\,D]\!] & X \mapsto (s : S) \times [\![D\,s]\!]\,X \\
[\![\text{'}\mathsf{ind}\times\,D]\!]\,X & \mapsto X \times [\![D]\!]\,X
\end{array}
$$

Figure 3. Universe of Descriptions

3.3 Examples

We begin with the natural numbers, now working in the high-level expression language of Section 2.3, exploiting type propagation.

$$
\begin{array}{l}
\mathsf{NatD} : \mathsf{Desc}^n \\
\mathsf{NatD} \mapsto \text{'}\Sigma\,\#[\text{'}\mathsf{zero}\ \text{'}\mathsf{suc}]\,[\text{'}1 \quad (\text{'}\mathsf{ind}\times\ \text{'}1)]
\end{array}
$$

Let us explain its construction. First, we use 'Σ to give a choice between the 'zero and 'suc constructors. What follows depends on this choice, so we write the function computing the rest of the description in tuple notation. In the 'zero case, we reach the end of the description. In the 'suc case, we attach one recursive argument and close the description. Translating the Σ to a binary sum, we have effectively described the functor:

$$\mathsf{NatD}\,Z \mapsto 1 + Z$$

Correspondingly, we can see the injections to the sum:

$$[\text{'}\mathsf{zero}] : [\![\mathsf{NatD}]\!]\,Z \qquad [\text{'}\mathsf{suc}\,(z : Z)] : [\![\mathsf{NatD}]\!]\,Z$$

The pattern functor for lists needs but a small change:

$$
\begin{array}{l}
\mathsf{ListD} : \textsc{Set}^n \to \mathsf{Desc}^n \\
\mathsf{ListD}\,X \mapsto \text{'}\Sigma\,\#[\text{'}\mathsf{nil}\ \text{'}\mathsf{cons}]\,[\text{'}1 \quad (\text{'}\Sigma\,X\,\lambda\_.\,\text{'}\mathsf{ind}\times\ \text{'}1)]
\end{array}
$$

The 'suc constructor becomes 'cons, taking an X followed by a recursive argument. This code describes the following functor:

$$\mathsf{ListD}\,X\,Z \mapsto 1 + X \times Z$$

Of course, we are not limited to one recursive argument. Here are the node-labelled binary trees:

$$
\begin{array}{l}
\mathsf{TreeD} : \textsc{Set}^n \to \mathsf{Desc}^n \\
\mathsf{TreeD}\,X \mapsto \text{'}\Sigma\,\#[\text{'}\mathsf{leaf}\ \text{'}\mathsf{node}] \\
\qquad\qquad [\text{'}1 \quad (\text{'}\mathsf{ind}\times\,(\text{'}\Sigma\,X\,\lambda\_.\,\text{'}\mathsf{ind}\times\ \text{'}1))]
\end{array}
$$

Again, we are one evolutionary step away from ListD. However, instead of a single call to the induction code, we add another. The interpretation of this code corresponds to the following functor:

$$\mathsf{TreeD}\,X\,Z \mapsto 1 + Z \times X \times Z$$

From the examples above, we observe that datatypes are defined by a 'Σ whose first argument enumerates the constructors. We call codes fitting this pattern *tagged* descriptions. Again, this is a clear reminder of the sum-of-products style. Any description can be forced into this style with a singleton constructor set. We characterise tagged descriptions thus:

$$
\begin{array}{l}
\mathsf{TagDesc}^n : \textsc{Set}^{n+1} \\
\mathsf{TagDesc}^n \mapsto (E : \mathsf{En}) \times (\pi\,E\,\lambda\_.\,\mathsf{Desc}^n)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{de} : \mathsf{TagDesc}^n \to \mathsf{Desc}^n \\
\mathsf{de} \mapsto \wedge\lambda E.\,\lambda D.\,\text{'}\Sigma\,\#E\,(\mathsf{switch}\,E\,(\lambda\_.\,\mathsf{Desc}^n)\,D)
\end{array}
$$

It is not such a stretch to expect that the familiar datatype declaration might desugar to the *definitions* of a tagged description.

3.4 The least fixpoint

So far, we have built pattern functors with our Desc universe. Being polynomial functors, they all admit a least fixpoint, which we now construct by *tying the knot*: the element type abstracted by the functor is now instantiated recursively:

$$\frac{\Gamma \vdash D : \mathsf{Desc}^n}{\Gamma \vdash \mu D : \mathrm{SET}^n} \qquad \frac{\Gamma \vdash D : \mathsf{Desc}^n \quad \Gamma \vdash d : [\![D]\!]\,(\mu D)}{\Gamma \vdash \mathsf{con}\, d : \mu D}$$

Tagged descriptions are very common, so we abbreviate:

$$\mu^+ : \mathsf{TagDesc}^n \to \mathrm{SET}^n \qquad \mu^+\, T \mapsto \mu(\mathsf{de}\, T)$$

We can now build datatypes and their elements, e.g.:

$$\mathsf{Nat} \mapsto \mu^+\,[[\text{'}\mathsf{zero}\ \text{'}\mathsf{suc}], [\text{'}1 \quad (\text{'}\mathsf{ind}\times\ \text{'}1)]] : \mathrm{SET}^n$$
$$\mathsf{con}\,[\text{'}\mathsf{zero}] : \mathsf{Nat} \qquad \mathsf{con}\,[\text{'}\mathsf{suc}\,(n : \mathsf{Nat})] : \mathsf{Nat}$$

But how shall we compute with our data? We should expect an elimination principle. Following a categorical intuition, we might provide the 'fold', or 'iterator', or 'catamorphism':

$$\mathsf{cata} : (D : \mathsf{Desc}^n)(T : \mathrm{SET}^n) \to ([\![D]\!]\, T \to T) \to \mu D \to T$$

However, iteration is inadequate for *dependent* computation. We need *induction* to write functions whose type depends on inductive data. Following Benke et al. [2003], we adopt the following:

$$\begin{array}{l}\mathsf{ind} : (D : \mathsf{Desc}^n)(P : \mu D \to \mathrm{SET}^k) \to \\ \quad ((d : [\![D]\!]\,(\mu D)) \to \mathsf{All}\, D\,(\mu D)\, P\, d \to P(\mathsf{con}\, d)) \to \\ \quad (x : \mu D) \to P x \\ \mathsf{ind}\, D\, P\, m\, (\mathsf{con}\, d) \mapsto m\, d\, (\mathsf{all}\, D\,(\mu D)\, P\, (\mathsf{ind}\, D\, P\, m)\, d)\end{array}$$

Here, $\mathsf{All}\, D\, X\, P\, d$ states that $P : X \to \mathrm{SET}^k$ holds for every subobject $x : X$ in D, and $\mathsf{all}\, D\, X\, P\, p\, d$ is a 'dependent map', applying some $p : (x : X) \to P x$ to each x contained in d. The definition (including an extra case, introduced soon) is in Figure 4.[2] So, ind is our first operation generic over descriptions, albeit hardwired. Any datatype we define comes with induction.

Note that the very same functors $[\![D]\!]$ also admit greatest fixpoints: we have indeed implemented coinductive types this way, but that is another story.

3.5 Extending type propagation

We now have low level machinery to build and manipulate inductive types. Let us apply cosmetic surgery to reduce the syntactic overhead. We extend type checking of expressions:

$$\frac{\Gamma \Vdash \#E \ni \text{'}c \triangleright n \quad \Gamma \Vdash [\![D\, n]\!]\,(\mu(\text{'}\Sigma\, \#E\, D)) \ni [\vec{t}] \triangleright t'}{\Gamma \Vdash \mu(\text{'}\Sigma\, \#E\, D) \ni \text{'}c\vec{t} \triangleright \mathsf{con}\,[n, t']}$$

Here $\text{'}c\vec{t}$ denotes a tag 'applied' to a sequence of arguments, and $[\vec{t}]$ that sequence's repackaging as a right-nested tuple. Now we can just write data directly.

$$\text{'}\mathsf{zero} : \mathsf{Nat} \qquad \text{'}\mathsf{suc}\,(n : \mathsf{Nat}) : \mathsf{Nat}$$

Once again, the type explains the legible presentation, as well as the low-level representation.

We may also simplify appeals to induction by type propagation, as we have done with functions from pairs and enumerations.

$$\frac{\Gamma \Vdash (d : [\![D]\!]\,(\mu D)) \to \mathsf{All}\, D\,(\mu D)\,(\lambda_{\mu D} x.\, P)\, d \to P[\mathsf{con}\, d/x] \quad \ni f \triangleright f'}{\Gamma \Vdash (x : \mu D) \to P \ni \circlearrowleft f \triangleright \mathsf{ind}\, D\,(\lambda_{\mu D} x.\, P) f'}$$

[2] To pass the termination checker, we had to inline the definition of all into ind in our Agda model. A simulation argument shows that the definition presented here terminates if the inlined version does. Hence, although not directly structural, this definition is indeed terminating.

This abbreviation is no substitute for the dependent pattern matching to which we are entitled in a high-level language built on top of this theory [Goguen et al. 2006]. It does at least make 'assembly language' programming mercifully brief, albeit hieroglyphic.

$$\begin{array}{l}\mathsf{plus} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{plus} \mapsto \circlearrowleft \wedge [(\lambda\_.\, \lambda\_.\, \lambda y.\, y) \quad (\lambda\_.\, \wedge \lambda h.\, \lambda\_.\, \lambda y.\, \text{'}\mathsf{suc}\,(h\, y))]\end{array}$$

This concludes our introduction to the universe of datatype descriptions. We have encoded sum-of-products datatypes from the simply-typed world as data and equipped them with computation. We have also made sure to hide the details by type propagation.

4. Levitating the Universe of Descriptions

In this section, we will fulfil our promises and show how we implement the signatures, first for the enumerations, and then for the codes of the Desc^n universe. Persuading these programs to perform was a perilous pedagogical peregrination for the protagonist. Our method was indeed to hardwire constants implementing the signatures specified above, in the first instance, but then attempt to replace them, step by step, with *definitions*: "Is $2+2$ still 4?", "No, it's a loop!". But we did find a way, so now we hope to convey to you the dizzy feeling of levitation, without the falling.

4.1 Implementing finite enumerations

In Section 2.2, we specified the finite sets of tags. We are going to implement (at every universe level) the En type former and its constructors. Recall:

$$\mathsf{En} : \mathrm{SET}^n \qquad \mathsf{nE} : \mathsf{En} \qquad \mathsf{cE}\,(t : \mathsf{Tag})\,(E : \mathsf{En}) : \mathsf{En}$$

The nE and cE constructors are just the 'nil' and 'cons' or ordinary lists, with elements from Tag. Therefore, we implement:

$$\mathsf{En} \mapsto \mu(\mathsf{ListD}\ \mathsf{Tag}) \qquad \mathsf{nE} \mapsto \text{'}\mathsf{nil} \qquad \mathsf{cE}\, t\, E \mapsto \text{'}\mathsf{cons}\, t\, E$$

Let us consider the consequences. We find that the type theory does not need a special type former En, or special constructors nE and cE. Moreover, the $\pi\, E\, P$ operator, computing tuple types of Ps by recursion on E need not be hardwired: we can just use the generic ind operator, as we would for any ordinary program.

Note, however, that the universe decoder $\#E$ *is* hardwired, as are the primitive 0 and $1+$ that we use for low-level values, and indeed the switch operator. We cannot dispose of data altogether! We have, however, gained the ordinariness of the enumeration *codes*, and hence of generic programs which manipulate them. Our next step is similar: we are going to condense the entire naming scheme of datatypes *into itself*.

4.2 Implementing descriptions

The set of codes, Desc, is already some sort of datatype; as with En, we ought to be able to describe it, coding of Desc^n in Desc^{n+1}, spiralling upwards. Hence, this code would be a first-class citizen, born with the generic equipment of datatypes.

4.2.1 First attempt

Our first attempt gets stuck quite quickly:

$$\mathsf{DescD}^n : \mathsf{Desc}^{n+1}$$
$$\mathsf{DescD}^n \mapsto \mathsf{de}\left[\begin{bmatrix}\text{'}1 \\ \text{'}\Sigma \\ \text{'}\mathsf{ind}\times\end{bmatrix}, \begin{bmatrix}\text{'}1 \\ \text{'}\Sigma\ \mathrm{SET}^n\ \lambda S.\ \{?\} \\ \text{'}\mathsf{ind}\times\ \text{'}1\end{bmatrix}\right]$$

Let us explain where we stand. Much as we have done so far, we first offer a constructor choice from '1, 'Σ, and 'ind×. You may notice that the 'tagged' notation we have used for the Desc^n constructors now fits the facts: these were actually the tags we are defining. For '1, we immediately reach the end of the description.

$$
\begin{array}{ll}
\mathsf{All}: & (D:\mathsf{Desc}^n)(X:\mathrm{SET}^n)(P:X\to\mathrm{SET}^k) \\
& (xs:[\![D]\!]\,X)\to\mathrm{SET}^k \\
\mathsf{All}\,\text{'}1 \quad XP[] & \mapsto 1 \\
\mathsf{All}\,(\text{'}\Sigma\,S\,D) \quad XP\,[s,d] & \mapsto \mathsf{All}\,(D\,s)\,XP\,d \\
\mathsf{All}\,(\text{'}\mathsf{ind}\times\,D) \quad XP\,[x,d] & \mapsto Px\times\mathsf{All}\,D\,XP\,d \\
\mathsf{All}\,(\text{'}\mathsf{hind}\times\,H\,D)\,XP\,[f,d] & \mapsto ((h:H)\to P\,(f\,h))\times\mathsf{All}\,D\,XP\,d
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{all}: & (D:\mathsf{Desc}^n)(X:\mathrm{SET}^n)(P:X\to\mathrm{SET}^k) \\
& (p:(x:X)\to Px)(xs:[\![D]\!]\,X)\to\mathsf{All}\,D\,XP\,xs \\
\mathsf{all}\,\text{'}1 \quad XPp[] & \mapsto [] \\
\mathsf{all}\,(\text{'}\Sigma\,S\,D) \quad XPp\,[s,d] & \mapsto \mathsf{all}\,(D\,s)\,XPp\,d \\
\mathsf{all}\,(\text{'}\mathsf{ind}\times\,D) \quad XPp\,[x,d] & \mapsto [p\,x,\mathsf{all}\,D\,XPp\,d] \\
\mathsf{all}\,(\text{'}\mathsf{hind}\times\,H\,D)\,XPp\,[f,d] & \mapsto [\lambda h.p\,(f\,h),\mathsf{all}\,D\,XPp\,d]
\end{array}
$$

Figure 4. Defining and collecting inductive hypotheses

For 'ind×, there is a single recursive argument. Describing 'Σ is problematic. Recall the specification of 'Σ:

$$\text{'}\Sigma\,(S:\mathrm{SET}^n)\,(D:S\to\mathsf{Desc}^n):\mathsf{Desc}^n$$

So, we first pack a SET^n, S, as well we might when working in Desc^{n+1}. We should then like a recursive argument *indexed* by S, but that is an *exponential*, and our presentation so far delivers only sums-of-products. To code our universe, we must first enlarge it!

4.2.2 Second attempt

In order to capture a notion of higher-order induction, we add a code 'hind× that takes an indexing set H. This amounts to give a recursive subobject for each element of H.

$$\text{'}\mathsf{hind}\times\,(H:\mathrm{SET}^n)\,(D:\mathsf{Desc}^n):\mathsf{Desc}^n$$
$$[\![\text{'}\mathsf{hind}\times\,H\,D]\!]\,X\mapsto(H\to X)\times[\![D]\!]\,X$$

Note that up to isomorphism, 'ind× is subsumed by 'hind× 1. However, the apparent duplication has some value. Unlike its counterpart, 'ind× is first-order: we prefer not to demand dummy functions from 1 in ordinary data, e.g. 'suc (λ_. n). It is naïve to imagine that up to isomorphism, any representation of data will do. First-order representations are finitary by construction, and thus admit a richer, componentwise decidable equality than functions may in general possess.[3]

We are now able to describe our universe of datatypes:

$$\mathsf{DescD}^n:\mathsf{Desc}^{n+1}$$
$$\mathsf{DescD}^n\mapsto\mathsf{de}\left[\begin{bmatrix}\text{'}1\\ \text{'}\Sigma\\ \text{'}\mathsf{ind}\times\\ \text{'}\mathsf{hind}\times\end{bmatrix},\begin{bmatrix}\text{'}1\\ \text{'}\Sigma\,\mathrm{SET}^n\,\lambda S.\,\text{'}\mathsf{hind}\times\,S\,\text{'}1\\ \text{'}\mathsf{ind}\times\,\text{'}1\\ \text{'}\Sigma\,\mathrm{SET}^n\,\lambda\_.\,\text{'}\mathsf{ind}\times\,\text{'}1\end{bmatrix}\right]$$

The '1 and 'ind× cases remain unchanged, as expected. We successfully describe the 'Σ case via the higher-order induction, branching on S. The 'hind× case just packs a SET^n with a recursive argument.

At a first glance, we have achieved our goal. We have described the codes of the universe of descriptions. The fixpoint of $[\![\mathsf{DescD}^n]\!]$ is a datatype just like Desc^n, in SET^{n+1}. Might we be so bold as to take $\mathsf{Desc}^n\mapsto\mu\mathsf{DescD}^n$ as the levitating definition? If we do, we shall come down with a bump! To complete our levitation, just as in the magic trick, requires hidden assistance. Let us explain the problem and reveal the 'invisible cable' which fixes it.

4.2.3 Final move

The definition $\mathsf{Desc}^n\mapsto\mu\mathsf{DescD}^n$ is circular, but the offensive recursion is concealed by a prestidigitation.

[3] E.g., extensionally, there is one function in $\#[]\to\mathsf{Nat}$; intensionally, there is a countable infinitude which it is dangerous to identify definitionally.

Expanding $\mathsf{de}-$ and propagating types as in Figure 2 reveals the awful truth:

$$\mathsf{Desc}^n\mapsto\mu(\text{'}\Sigma\#[\text{'}1\ \text{'}\Sigma\ \text{'}\mathsf{ind}\times\ \text{'}\mathsf{hind}\times]\ \mathsf{switch}\,[\text{'}1\ \text{'}\Sigma\ \text{'}\mathsf{ind}\times\ \text{'}\mathsf{hind}\times]\,(\lambda\_.\,\mathsf{Desc}^{n+1})\begin{bmatrix}\text{'}1\\ \text{'}\Sigma\,\mathrm{SET}^n\,\lambda S.\,\text{'}\mathsf{hind}\times\,S\,\text{'}1\\ \text{'}\mathsf{ind}\times\,\text{'}1\\ \text{'}\Sigma\,\mathrm{SET}^n\,\lambda\_.\,\text{'}\mathsf{ind}\times\,\text{'}1\end{bmatrix})$$

The recursion shows up only because we must specify the return type of the general-purpose switch, and it is computing a Desc^{n+1}! Although type propagation allows us to hide this detail *when defining a function*, we cannot readily suppress this information and check types when switch is fully applied.

We are too close to give up now. If only we did not need to supply that return type, especially when we know what it must be! We eliminate the recursion by *specialising* switch:

$$\mathsf{switchD}:(E:\mathsf{En})\to(\pi\,E\,\lambda\_.\,\mathsf{Desc}^m)\to\#E\to\mathsf{Desc}^m$$

The magician's art rests here, in this extension. We conceal it behind a type propagation rule for switchD which we apply with higher priority than for switch in general.

$$\frac{\Gamma\Vdash\pi\,E\,\lambda_{\#E}x.\,\mathsf{Desc}^m\ni[\vec{t}]\triangleright t'}{\Gamma\Vdash\#E\to\mathsf{Desc}^m\ni[\vec{t}]\triangleright\mathsf{switchD}\,E\,t'}$$

As a consequence, our definition above now propagates without introducing recursion. Of course, by pasting together the declaration of Desc^n and its internal copy, we have made it appear in its own type. Hardwired as a trusted *fait accompli*, this creates no regress, although one must assume the definition to recheck it.

Our Agda model does not formalise the switchD construction. Instead, we exhibit the isomorphism between declared and encoded descriptions. Here, switchD lets us collapse this isomorphism, operationally identifying defined and coded descriptions.

There are other ways to achieve a sufficient specialisation to avoid a recursive code, e.g., extending Desc^n with specialised codes for *finite* sums and products, pushing the switch into the interpretation of codes, rather than the code itself. Here, we prefer not to add codes to Desc^n which are otherwise unmotivated.

We have levitated Desc at every level. Beyond its pedagogical value, this exercise has several practical outcomes. First, it confirms that each Desc universe is just plain data. As any piece of data, it can therefore be inspected and manipulated. Moreover, it is expressed in a Desc universe. As a consequence, it is equipped, for free, with an induction principle. So, our ability to inspect and program with Desc is not restricted to a meta-language: we have the necessary equipment to program with *data*, so we can program over datatypes. *Generic programming is just programming*.

4.3 The generic catamorphism

In Section 3.4, we hardwired a dependent induction principle, but sometimes, iteration suffices. Let us construct the catamorphism.

We proceed by induction on the *data* in μD: the non-dependent return type T is readily propagated. Given a node xs and the in-

duction hypotheses, the method ought to build an element of T. Provided that we know how to make an element of $[\![D]\!]\,T$, this step will be performed by the algebra f. Let us take a look at this jigsaw:

$$\begin{array}{l}\mathsf{cata} : (D : \mathsf{Desc})(T : \mathrm{SET}) \to ([\![D]\!]\,T \to T) \to \mu D \to T \\ \mathsf{cata}\,D\,T\,f \mapsto \circlearrowleft \lambda xs.\,\lambda hs.f\ \{?\}\end{array}$$

The hole remains: we have $xs : [\![D]\!]\,\mu D$ and $hs : \mathsf{All}\,D\,\mu D\,(\lambda\_.\,T)\,xs$ to hand, and we need a $[\![D]\!]\,T$. Now, xs has the right shape, but its components have the wrong type. However, for each such component, hs holds the corresponding value in T. We need a function to replace the former with the latter: this pattern matching sketch yields an induction on D. We fill the hole with $\mathsf{replace}\,D\,(\mu D)\,T\,xs\,hs$.

$$\begin{array}{lll}\mathsf{replace} : (D : \mathsf{Desc})(X, Y : \mathrm{SET}) & & \\ \qquad (xs : [\![D]\!]\,X) \to \mathsf{All}\,D\,X\,(\lambda\_.\,Y)\,xs \to [\![D]\!]\,Y & & \\ \mathsf{replace}\ \text{'}1 & X\,Y\,[] \quad [] & \mapsto [] \\ \mathsf{replace}\ (\text{'}\Sigma\,S\,D) & X\,Y\,[s,d]\ d' & \mapsto [s, \mathsf{replace}\,(D\,s)\,X\,Y\,d\,d'] \\ \mathsf{replace}\ (\text{'}\mathsf{ind}\times\,D) & X\,Y\,[x,d]\ [y,d'] & \mapsto [y, \mathsf{replace}\,D\,X\,Y\,d\,d'] \\ \mathsf{replace}\ (\text{'}\mathsf{hind}\times\,H\,D) & X\,Y\,[f,d]\ [g,d'] & \mapsto [g, \mathsf{replace}\,D\,X\,Y\,d\,d']\end{array}$$

We have shown how to derive a generic operation, cata, from a pre-existing generic operation, ind, by manipulating descriptions as data: the catamorphism is just a function taking each Desc value to a datatype specific operation. This is polytypic programming, as in PolyP [Jansson and Jeuring 1997], made ordinary.

4.4 The generic free monad

In this section, we try a more ambitious generic operation. Given a functor—a signature of operations represented as a tagged description—we build its free monad, extending the signature with variables and substitution. Let us recall this construction in, say, Haskell. Given a functor `f`, the free monad over `f` is given thus:

```
data FreeMonad f x = Var x | Op (f (FreeMonad f x))
```

Provided `f` is an instance of `Functor`, we may take `Var` for `return` and use `f`'s `fmap` to define »= as substitution.

Being an inductive type, FreeMonad arises by a pattern functor:

$$\mathsf{FreeMonadD}\,F\,X\,Z \mapsto X + F\,Z$$

Our construction takes the functor as a tagged description, and given a set X of variables, computes the tagged description of the free monad pattern functor.

$$\begin{array}{l}\_^* : \mathsf{TagDesc} \to \mathrm{SET} \to \mathsf{TagDesc} \\ [E,D]^*\,X \mapsto [[\text{'}\mathsf{var}, E], [\text{'}\Sigma\,X\,\text{'}1, D]]\end{array}$$

We simply add a constructor, 'var, making its arguments $\text{'}\Sigma\,X\,\text{'}1$—just an element of X. E and D stay put, leaving the other constructors unchanged. Unfolding the interpretation of this definition, we find an extended sum, corresponding to the $X+$ in FreeMonadD. Taking the fixpoint ties the knot and we have our data.

Now we need the operations. As expected, $\lambda x.\,\text{'}\mathsf{var}\,x$ plays the rôle of *return*, making variables terms. Meanwhile, *bind* is indeed *substitution*, which we now implement generically, making use of cata. Let us write the type, and start filling in the blanks:

$$\begin{array}{ll}\mathsf{subst} : & (D : \mathsf{TagDesc})(X, Y : \mathrm{SET}) \to (X \to \mu^+\,(D^*\,Y)) \to \\ & \mu^+\,(D^*\,X) \to \mu^+\,(D^*\,Y) \\ \mathsf{subst}\,D\,X\,Y\,\sigma \mapsto \mathsf{cata}\,(\mathsf{de}\,(D^*\,X))\,(\mu^+\,(D^*\,Y))\ \{?\}\end{array}$$

We are left with implementing the algebra of the catamorphism. Its role is to catch appearances of $\text{'}\mathsf{var}\,x$ and replace them by $\sigma\,x$. This corresponds to the following definition:

$$\begin{array}{ll}\mathsf{apply} : & (D : \mathsf{TagDesc})(X, Y : \mathrm{SET}) \to (X \to \mu^+\,(D^*\,Y)) \to \\ & [\![\mathsf{de}\,(D^*\,X)]\!]\,(\mu^+\,(D^*\,Y)) \to \mu^+\,(D^*\,Y) \\ \mathsf{apply}\,D\,X\,Y\,\sigma\,[\text{'}\mathsf{var}\,x] \mapsto \sigma\,x & \\ \mathsf{apply}\,D\,X\,Y\,\sigma\,[c,xs] \mapsto \mathsf{con}\,[c,xs] & \end{array}$$

Object	Role	Status
En	Build finite sets	Levitated
Desc	Describe pattern functors	Levitated
$[\![\_]\!]$	Interpret descriptions	Hardwired
μ, con	Define, inhabit fixpoints	Hardwired
ind, All, all	Induction principle	Hardwired

Table 1. Summary of constructions on Descriptions

We complete the hole with $\mathsf{apply}\,D\,X\,Y\,\sigma$. Every tagged description can be seen as a signature of operations: we can uniformly add a notion of variable, building a new type from an old one, then providing the substitution structure.

4.5 Skyhooks all the way up?

In this section, we have seen how to *levitate* descriptions. Although our theory, as presented here, takes $\mathrm{SET} : \mathrm{SET}$, our annotations indicate how a stratified theory could code each level from above. We do not rely on the paradoxical nature of $\mathrm{SET} : \mathrm{SET}$ to flatten the hierarchy of descriptions and fit large inside small. We shall now be more precise about what we have done.

Let us first clarify the status of the implementation. The kit for making datatypes is presented in Table 1. For each operation, we describe its role and its status, making clear which components are self-described and which ones are actually implemented.

In a stratified system, the 'self-encoded' nature of Desc appears only in a set polymorphic sense: the principal type of the encoded description generalises to the type of Desc itself. We encode this much in our set polymorphic model in Agda and in our Coq model, crucially relying on typical ambiguity [Harper and Pollack]. We step outside current technology only to replace the declared Desc with its encoding.

Even this last step, we can approximate within a standard predicative hierarchy. Fix a top level, perhaps 42. We may start by declaring $\mathsf{Desc}^{42} : \mathrm{SET}^{43}$. We can then construct $\mathsf{DescD}^{41} : \mathsf{Desc}^{42}$ and thus acquire an encoded Desc^{41}. Although Desc^{41} is encoded, not declared, it includes the relevant descriptions, including DescD^{40}. We can thus build the tower of descriptions down to Desc^{0}, encoding every level below the top. Description of descriptions forms a 'spiral', rather than a circle. We have modelled this process exactly in Agda, without any appeal to dependent pattern matching, induction-recursion, or set polymorphism. All it takes to build such a sawn-off model of encodings is inductive definition and a cumulative predicative hierarchy of set levels.

5. A Universe of Inductive Families

So far, we have explored the realm of inductive types, building on intuition from ML-like datatypes, using type dependency as a descriptive tool in Desc and its interpretation. Let us now make dependent types the object as well as the means of our study.

Dependent datatypes provide a way to work at higher level of precision *a priori*, reducing the sources of failure we might otherwise need to manage. For the perennial example, consider *vectors*—lists indexed by length. By making length explicit in the type, we can prevent hazardous operations (the type of 'head' demands vectors of length $\text{'}\mathsf{suc}\,n$) and offer stronger guarantees (pointwise addition of n-vectors yields an n-vector).

However, these datatypes are not *individually* inductive. For instance, we have to define the whole *family* of vectors mutually, in one go. In dependently typed languages, the basic grammar of datatypes is that of inductive families. To capture this grammar, we must account for *indexing*.

5.1 The universe of indexed descriptions

We presented the Desc universe as a grammar of strictly positive endofunctors on SET and developed inductive types by taking a fixpoint. To describe inductive families indexed by some $I : \mathrm{SET}$, we play a similar game with endofunctors on the category SET^I, families of sets $X, Y : I \to \mathrm{SET}$ for objects, and for morphisms, families of functions in $X \dot{\to} Y$, defined pointwise:

$$X \dot{\to} Y \mapsto (i : I) \to X\, i \to Y\, i$$

An *indexed functor* in $\mathrm{SET}^I \to \mathrm{SET}^J$ has the flavour of a device driver, characterising 'responses' to a given request in J where we may in turn make 'subrequests' at indices chosen from I. When we use indexed functors to define inductive families of datatypes, I and J coincide: we explain how to make a node fit a given index, including subnodes at chosen indices. E.g., if we are asked for a vector of length 3, we choose to ask in turn for a tail of length 2.

To code up valid notions of response to a given request, we introduce IDesc and its interpretation:

$$\begin{array}{l}\mathsf{IDesc}\,(I : \mathrm{SET}) : \mathrm{SET} \\ [\![\_]\!] :_{(I:\mathrm{SET})\to} \mathsf{IDesc}\, I \to (I \to \mathrm{SET}) \to \mathrm{SET}\end{array}$$

An $\mathsf{IDesc}\, I$ specifies just *one* response, but a request-to-response *function*, $R : I \to \mathsf{IDesc}\, I$, yields a strictly positive endofunctor

$$\lambda X.\, \lambda i.\, [\![R\, i]\!]_I\, X : \mathrm{SET}^I \to \mathrm{SET}^I$$

whose fixpoint we then take:

$$\frac{\Gamma \vdash I : \mathrm{SET} \qquad \Gamma \vdash R : I \to \mathsf{IDesc}\, I}{\Gamma \vdash \mu_I R : I \to \mathrm{SET}}$$

$$\frac{\begin{array}{ll}\Gamma \vdash I : \mathrm{SET} & \Gamma \vdash R : I \to \mathsf{IDesc}\, I \\ \Gamma \vdash i : I & \Gamma \vdash x : [\![R\, i]\!]_I\, (\mu_I R)\end{array}}{\Gamma \vdash \mathsf{con}\, x : \mu_I R\, i}$$

We define the IDesc grammar in Figure 6, delivering only *strictly positive* families. As well as indexing our descriptions, we have refactored a little, adopting a more compositional algebra of codes, where Desc is biased towards the right-nested tuples. We now have $\text{'}\mathsf{var}\, i$ for recursive 'subrequests' at a chosen index i, with tupling by right-associative $\text{'}\times$ and higher-order branching by $\text{'}\Pi$. Upgrade your old Desc to a trivially indexed $\mathsf{IDesc}\,1$ as follows!

$$\begin{array}{ll}\mathsf{upgrade} : \mathsf{Desc} & \to \mathsf{IDesc}\,1 \\ \mathsf{upgrade}\ \text{'}1 & \mapsto \text{'}\mathsf{k}\,1 \\ \mathsf{upgrade}\ (\text{'}\Sigma\, S\, D) & \mapsto \text{'}\Sigma\, S\, \lambda s.\, \mathsf{upgrade}\ (D\, s) \\ \mathsf{upgrade}\ (\text{'}\mathsf{ind}\times\, D) & \mapsto \text{'}\mathsf{var}\,[]\,\text{'}\times\, \mathsf{upgrade}\, D \\ \mathsf{upgrade}\ (\text{'}\mathsf{hind}\times\, H\, D) & \mapsto (\text{'}\Pi\, H\, \lambda\_.\, \text{'}\mathsf{var}\,[])\,\text{'}\times\, \mathsf{upgrade}\, D\end{array}$$

To deliver induction for indexed datatypes, we need the 'holds everywhere' machinery. We present AllI and allI in Figure 5, with a twist—where Desc admits the all construction, IDesc is *closed* under it! The AllI operator for a description indexed on I is strictly positive in turn, and has a description indexed on some $(i : I) \times X\, i$. Induction on indexed descriptions is then hardwired thus:

$$\begin{array}{ll}\mathsf{indI} : & {}_{(I:\mathrm{SET})\to}(R : I \to \mathsf{IDesc}\, I)(P : ((i : I) \times \mu_I R\, i) \to \mathrm{SET}) \to \\ & ((i : I)(xs : [\![R\, i]\!]_I\, (\mu_I R)) \to \\ & [\![\mathsf{AllI}\, (R\, i)\, (\mu_I R)\, xs]\!]\, P \to P\, [i, \mathsf{con}\, xs]) \to \\ & (i : I)(x : \mu_I R\, i) \to P\, [i, x] \\ \mathsf{indI}\, R\, P\, m\, i\, (\mathsf{con}\, xs) \mapsto & m\, i\, xs\, (\mathsf{allI}\, R\, i\, (\mu_I R)\, P\, (\wedge\lambda i.\, \lambda xs.\, \mathsf{indI}\, R\, P\, m)\, xs)\end{array}$$

The generic catamorphism, catal, is constructed from indI as before. Its type becomes more elaborated, to deal with the indexing:

$$\begin{array}{l}\mathsf{catal} : (I : \mathrm{SET})(R : I \to \mathsf{IDesc}\, I) \\ \qquad (T : I \to \mathrm{SET}) \to ((i : I) \to [\![R\, i]\!]\, T \to T\, i) \to \mu_I R \dot{\to} T\end{array}$$

$$\begin{array}{ll}\mathsf{IDesc}\,(I : \mathrm{SET}) & : \mathrm{SET} \\ \text{'}\mathsf{var}\,(i : I) & : \mathsf{IDesc}\, I \\ \text{'}\mathsf{k}\,(A : \mathrm{SET}) & : \mathsf{IDesc}\, I \\ (D : \mathsf{IDesc}\, I)\,\text{'}\times (D : \mathsf{IDesc}\, I) & : \mathsf{IDesc}\, I \\ \text{'}\Sigma\,(S : \mathrm{SET})\,(D : S \to \mathsf{IDesc}\, I) & : \mathsf{IDesc}\, I \\ \text{'}\Pi\,(S : \mathrm{SET})\,(D : S \to \mathsf{IDesc}\, I) & : \mathsf{IDesc}\, I\end{array}$$

$$\begin{array}{lll}[\![\_]\!] :_{(I:\mathrm{SET})\to} \mathsf{IDesc}\, I \to (I \to \mathrm{SET}) \to \mathrm{SET} & & \\ [\![\text{'}\mathsf{var}\, i]\!]_I & X \mapsto & X\, i \\ [\![\text{'}\mathsf{k}\, K]\!]_I & X \mapsto & K \\ [\![D\,\text{'}\times D']\!]_I\, X & \mapsto & [\![D]\!]_I\, X \times [\![D']\!]_I\, X \\ [\![\text{'}\Sigma\, S\, D]\!]_I & X \mapsto & (s : S) \times [\![D\, s]\!]_I\, X \\ [\![\text{'}\Pi\, S\, D]\!]_I & X \mapsto & (s : S) \to [\![D\, s]\!]_I\, X\end{array}$$

Figure 6. Universe of indexed descriptions

5.2 Examples

Natural numbers: For basic reassurance, we upgrade NatD:

$$\begin{array}{l}\mathsf{upgrade\ NatD} : \mathsf{IDesc}\,1 \\ \mathsf{upgrade\ NatD} \mapsto \text{'}\Sigma\,(\#[\text{'}\mathsf{zero}\ \text{'}\mathsf{suc}])\,[(\text{'}\mathsf{k}\,1)\ (\text{'}\mathsf{var}\,[]\,\text{'}\times\, \text{'}\mathsf{k}\,1)]\end{array}$$

Note that trailing 1's keep our right-nested, []-terminated tuple structure, and with it our elaboration machinery. We can similarly upgrade any inductive type. Moreover, $\mathsf{IDesc}\, I$ can now code a bunch of mutually inductive types, if I enumerates the bunch [Paulin-Mohring 1996; Yakushev et al. 2009].

Indexed descriptions: Note that $\mathsf{IDesc}\, I$ is a plain inductive type, parametrised by I, but indexed trivially.

$$\begin{array}{l}\mathsf{IDescD} : (I : \mathrm{SET}) \to \mathsf{IDesc}\,1 \\ \mathsf{IDescD}\, I \mapsto \text{'}\Sigma \\ \qquad \#\begin{bmatrix}\text{'}\mathsf{var} \\ \text{'}\mathsf{k} \\ \text{'}\times \\ \text{'}\Sigma \\ \text{'}\Pi\end{bmatrix}\begin{bmatrix}(\text{'}\mathsf{k}\, I & \text{'}\times\, \text{'}\mathsf{k}\,1) \\ (\text{'}\mathsf{k}\, \mathrm{SET} & \text{'}\times\, \text{'}\mathsf{k}\,1) \\ (\text{'}\mathsf{var}\,[]\,\text{'}\times\, \text{'}\mathsf{var}\,[] & \text{'}\times\, \text{'}\mathsf{k}\,1) \\ (\text{'}\Sigma\, \mathrm{SET}\, \lambda S.\, (\text{'}\Pi\, S\, \lambda\_.\, \text{'}\mathsf{var}\,[])\,\text{'}\times\, \text{'}\mathsf{k}\,1) \\ (\text{'}\Sigma\, \mathrm{SET}\, \lambda S.\, (\text{'}\Pi\, S\, \lambda\_.\, \text{'}\mathsf{var}\,[])\,\text{'}\times\, \text{'}\mathsf{k}\,1)\end{bmatrix}\end{array}$$

Therefore, this universe is self-describing and can be levitated. As before, we rely on a special purpose switchID operator to build the finite function $[\ldots]$ without mentioning IDesc.

Vectors: So far, our examples live in $\mathsf{IDesc}\,1$, with no interesting indexing. Let us at least have vectors. Recall that the constructors $\text{'}\mathsf{vnil}$ and $\text{'}\mathsf{vcons}$ are defined only for $\text{'}\mathsf{zero}$ and $\text{'}\mathsf{suc}$ respectively:

$$\begin{array}{l}\mathbf{data}\ \mathsf{Vec}\,(X : \mathrm{SET}) : (i : \mathsf{Nat}) \to \mathrm{SET}\ \mathbf{where} \\ \quad \text{'}\mathsf{vnil} \quad : \mathsf{Vec}\, X\, \text{'}\mathsf{zero} \\ \quad \text{'}\mathsf{vcons} : {}_{(n:\mathsf{Nat})\to} X \to \mathsf{Vec}\, X\, n \to \mathsf{Vec}\, X\, (\text{'}\mathsf{suc}\, n)\end{array}$$

One way to code constrained datatypes is to appeal to a suitable notion of propositional equality == on indices. The constraints are expressed as 'Henry Ford' equations in the datatype. For vectors:

$$\begin{array}{l}\mathsf{VecD} : \mathrm{SET} \to \mathsf{Nat} \to \mathsf{IDesc}\,\mathsf{Nat} \\ \mathsf{VecD}\, X\, i \mapsto \text{'}\Sigma \\ \qquad \#\begin{bmatrix}\text{'}\mathsf{vnil} \\ \text{'}\mathsf{vcons}\end{bmatrix}\begin{bmatrix}(\text{'}\mathsf{k}\,(\text{'}\mathsf{zero} == i)) \\ (\text{'}\Sigma\, \mathsf{Nat}\, \lambda n.\, \text{'}\mathsf{k}\, X\,\text{'}\times\, \text{'}\mathsf{var}\, n\,\text{'}\times\, \text{'}\mathsf{k}\,(\text{'}\mathsf{suc}\, n == i))\end{bmatrix}\end{array}$$

You may choose $\text{'}\mathsf{vnil}$ for any index you like as long as it is $\text{'}\mathsf{zero}$; in the $\text{'}\mathsf{vcons}$ case, the length of the tail is given explicitly, and the index i must be one more. Our previous 1-terminated tuple types can now be seen as the trivial case of constraint-terminated tuple types, with elaboration supplying the witnesses when trivial.

In this paper, we remain anxiously agnostic about propositional equality. Any will do, according to conviction; many variations are popular. The homogeneous identity type used in Coq is ill-suited to dependent types, but its heterogeneous variant (forming equations

$$
\begin{array}{ll}
\mathsf{AllI} : {}_{(I:\mathrm{SET})\to}(D:\mathsf{IDesc}\,I)(X:I\to\mathrm{SET})\to & \mathsf{allI} : {}_{(I:\mathrm{SET})\to}(D:\mathsf{IDesc}\,I)(X:I\to\mathrm{SET})(P:((i:I)\times X\,i)\to\mathrm{SET})\to \\
\quad [\![D]\!]_I\,X\to\mathsf{IDesc}\,((i:I)\times X\,i) & \quad ((x:(i:I)\times X\,i)\to P\,x)\to(xs:[\![D]\!]_I\,X)\to[\![\mathsf{AllI}\,D\,X\,xs]\!]\,P \\
\mathsf{AllI}\,(\text{'var}\,i)\quad X\,x & \mapsto \text{'var}\,[i,x] \\
\mathsf{AllI}\,(\text{'k}\,K)\quad X\,k & \mapsto \text{'k}\,1 \\
\mathsf{AllI}\,(D\text{'}\times D')\,X\,[d,d'] & \mapsto \mathsf{AllI}\,D\,X\,d\text{'}\times\mathsf{AllI}\,D'\,X\,d' \\
\mathsf{AllI}\,(\text{'}\Sigma\,S\,D)\quad X\,[s,d] & \mapsto \mathsf{AllI}\,(D\,s)\,X\,d \\
\mathsf{AllI}\,(\text{'}\Pi\,S\,D)\quad X\,f & \mapsto \text{'}\Pi\,S\,\lambda s.\,\mathsf{AllI}\,(D\,s)\,X\,(f\,s)
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{allI}\,(\text{'var}\,i)\quad X\,P\,p\,x & \mapsto p\,[i,x] \\
\mathsf{allI}\,(\text{'k}\,K)\quad X\,P\,p\,k & \mapsto [] \\
\mathsf{allI}\,(D\text{'}\times D')\,X\,P\,p\,[d,d'] & \mapsto [\mathsf{allI}\,D\,X\,P\,p\,d,\mathsf{allI}\,D'\,X\,P\,p\,d'] \\
\mathsf{allI}\,(\text{'}\Sigma\,S\,D)\quad X\,P\,p\,[s,d] & \mapsto \mathsf{allI}\,(D\,s)\,X\,P\,p\,d \\
\mathsf{allI}\,(\text{'}\Pi\,S\,D)\quad X\,P\,p\,f & \mapsto \lambda a.\,\mathsf{allI}\,(D\,a)\,X\,P\,p\,(f\,a)
\end{array}
$$

Figure 5. Indexed induction predicates

regardless of type) allows the translation of pattern matching with structural recursion to indI [Goguen et al. 2006]. The extensional equality of Altenkirch et al. [2007] also sustains the translation.

However, sometimes, the equations are redundant. Looking back at Vec, we find that the equations constrain the choice of constructor and stored tail index retrospectively. But *inductive families need not store their indices* [Brady et al. 2003]! If we analyse the incoming index, we can tidy our description of Vec as follows:

$$
\begin{array}{lcl}
\mathsf{VecD}\,(X:\mathrm{SET}):\mathsf{Nat}\to\mathsf{IDesc}\,\mathsf{Nat} & & \\
\mathsf{VecD}\,X\,\text{'zero} & \mapsto & \text{'k}\,1 \\
\mathsf{VecD}\,X\,(\text{'suc}\,n) & \mapsto & \text{'k}\,X\,\text{'}\times\text{'var}\,n
\end{array}
$$

The constructors and equations have simply disappeared. A similar example is Fin (bounded numbers), specified by:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Fin} : (n:\mathsf{Nat})\to\mathrm{SET}\ \textbf{where} \\
\quad \text{'fz} : {}_{(n:\mathsf{Nat})\to}\mathsf{Fin}\,(\text{'suc}\,n) \\
\quad \text{'fs} : {}_{(n:\mathsf{Nat})\to}\mathsf{Fin}\,n\to\mathsf{Fin}\,(\text{'suc}\,n)
\end{array}
$$

In this case, we can eliminate equations but not constructors, since both 'fz and 'fs both target 'suc:

$$
\begin{array}{lcl}
\mathsf{FinD} : \mathsf{Nat}\to\mathsf{IDesc}\,\mathsf{Nat} & & \\
\mathsf{FinD}\,\text{'zero} & \mapsto & \text{'}\Sigma\,\#[]\,[] \\
\mathsf{FinD}\,(\text{'suc}\,n) & \mapsto & \text{'}\Sigma\,\#[\text{'fz}\,\text{'fs}]\,[(\text{'k}\,1)\,(\text{'var}\,n)]
\end{array}
$$

This technique of extracting information by case analysis on indices applies to descriptions exactly where Brady's 'forcing' and 'detagging' optimisations apply in compilation. They eliminate just those constructors, indices and constraints which are redundant even in *open* computation. In *closed* computation, where proofs can be trusted, all constraints are dropped.

Tagged indexed descriptions: Let us reflect this index analysis technique. We can divide a description of tagged indexed data in two: first, the constructors that do not depend on the index; then, the constructors that do. The non-dependent part mirrors the definition for non-indexed descriptions. The index-dependent part simply indexes the choice of constructors by I. Hence, by inspecting the index, it is possible to vary the 'menu' of constructors.

$$
\begin{array}{l}
\mathsf{TagIDesc}\,I\mapsto\mathsf{AlwaysD}\,I\times\mathsf{IndexedD}\,I \\
\mathsf{AlwaysD}\,I\ \mapsto(E:\mathsf{En})\times(i:I)\to\pi\,E\,\lambda\_.\,\mathsf{IDesc}\,I \\
\mathsf{IndexedD}\,I\mapsto(F:I\to\mathsf{En})\times(i:I)\to\pi\,(F\,i)\,\lambda\_.\,\mathsf{IDesc}\,I
\end{array}
$$

In the case of a tagged Vec, for instance, for the index 'zero, we would only propose the constructor 'nil. Similarly, for $\text{'suc}\,n$, we would only propose the constructor 'cons.

We write $\mathsf{de}\,D\,i$ to denote the $\mathsf{IDesc}\,I$ computed from the tagged indexed description D at index i. Its expansion is similar to the definition of de for tagged descriptions, except that it must also append the two parts. We again write $\mu^+_I\,D$ for $\mu_I(\mathsf{de}\,D)$.

Typed expressions: We are going to define a syntax for a small language with two types, natural numbers and booleans:

$$
\mathsf{Ty}\mapsto\#[\text{'nat}\,\text{'bool}]
$$

This language has values, conditional expression, addition and comparison. Informally, their types are:

$$
\begin{array}{ll}
\text{'val}\ : \mathsf{Val}\,ty\to ty & \text{'plus} : \text{'nat}\to\text{'nat}\to\text{'nat} \\
\text{'cond} : \text{'bool}\to ty\to ty\to ty & \text{'le}\ \ : \text{'nat}\to\text{'nat}\to\text{'bool}
\end{array}
$$

The function Val interprets object language types in the host language, so that arguments to 'val fit their expected type.

$$
\begin{array}{l}
\mathsf{Val} : \mathsf{Ty}\to\mathrm{SET} \\
\mathsf{Val}\,\text{'nat}\ \mapsto \mathsf{Nat} \\
\mathsf{Val}\,\text{'bool}\mapsto \mathsf{Bool}
\end{array}
$$

We take Nat and Bool to represent natural numbers and Booleans in the host language, equipped with addition $+_{\mathrm{H}}$ and comparison $\leq_{\mathrm{H}}$.

We express our syntax as a tagged indexed description, indexing over object language types Ty. We note that some constructors are always available, namely 'val and 'cond. On the other hand, 'plus and 'le constructors are index-dependent, with 'plus available just when building a 'nat, 'le just for 'bool. The code, below, reflects this intuition, with the first component uniformly offering 'val and 'cond, the second selectively offering 'plus or 'le.

$$
\begin{array}{l}
\mathsf{ExprD} : \mathsf{TagIDesc}\,\mathsf{Ty} \\
\mathsf{ExprD}\mapsto[\mathsf{ExprAD},\mathsf{ExprID}] \\
\mathsf{ExprAD} : \mathsf{AlwaysD}\,\mathsf{Ty} \\
\mathsf{ExprAD}\mapsto\left[\begin{bmatrix}\text{'val}\\ \text{'cond}\end{bmatrix},\lambda ty.\begin{bmatrix}\text{'k}\,(\mathsf{Val}\,ty) & \text{'}\times\text{'k}\,1\\ \text{'var}\,\text{'bool}\text{'}\times\text{'var}\,ty\text{'}\times\text{'var}\,ty\text{'}\times\text{'k}\,1\end{bmatrix}\right] \\
\mathsf{ExprID} : \mathsf{IndexedD}\,\mathsf{Ty} \\
\mathsf{ExprID}\mapsto\left[\begin{bmatrix}[\text{'plus}]\\ [\text{'le}]\end{bmatrix},\lambda\_.\,[\text{'var}\,\text{'nat}\text{'}\times\text{'var}\,\text{'nat}\text{'}\times\text{'k}\,1]\right]
\end{array}
$$

Given the syntax, let us supply the semantics. We implement an evaluator as a catamorphism:

$$
\begin{array}{l}
\mathsf{eval}_{\Downarrow} : (ty:\mathsf{Ty})\to\mu^+_{\mathsf{Ty}}\,\mathsf{ExprD}\,ty\to\mathsf{Val}\,ty \\
\mathsf{eval}_{\Downarrow}\,ty\,term\mapsto\mathsf{cata}_{\mathsf{Ty}}\,(\mathsf{de}\,\mathsf{ExprD})\,\mathsf{Val}\,\mathsf{eval}_{\downarrow}\,ty\,term
\end{array}
$$

To finish the job, we must supply the algebra which implements a single step of evaluation, given subexpressions evaluated already.

$$
\begin{array}{lllll}
\mathsf{eval}_{\downarrow} : (ty:\mathsf{Ty})\to[\![(\mathsf{de}\,\mathsf{ExprD})\,ty]\!]_{\mathsf{Ty}}\,\mathsf{Val}\to\mathsf{Val}\,ty & & & & \\
\mathsf{eval}_{\downarrow} & \_ & (\text{'val}\,x) & \mapsto x & \\
\mathsf{eval}_{\downarrow} & \_ & (\text{'cond}\,\text{'true}\,x\,\_) & \mapsto x & \\
\mathsf{eval}_{\downarrow} & \_ & (\text{'cond}\,\text{'false}\,\_\,y) & \mapsto y & \\
\mathsf{eval}_{\downarrow} & \text{'nat} & (\text{'plus}\,x\,y) & \mapsto x+_{\mathrm{H}}y & \\
\mathsf{eval}_{\downarrow} & \text{'bool} & (\text{'le}\,x\,y) & \mapsto x\leq_{\mathrm{H}}y &
\end{array}
$$

Hence, we have a type-safe syntax and a tagless interpreter for our language, in the spirit of Augustsson and Carlsson [1999], with help from the generic catamorphism. However, so far, we are only able to define and manipulate *closed* terms. Adding variables, it is possible to build and manipulate *open* terms, that is, terms in a context. We shall get this representation, for free, thanks to the *free indexed monad* construction.

5.3 Free indexed monad

In Section 4.4, we have built a free monad operation for simple descriptions. The process is similar in the indexed world. Namely, given an indexed functor, we derive the indexed functor coding its free monad:

$$\begin{array}{l} \_^* : \ {}_{(I:\mathrm{SET})\to}(R : \mathsf{TagIDesc}\, I)(X : I \to \mathrm{SET}) \to \mathsf{TagIDesc}\, I \\ [E,F]^*_I\, R \mapsto \left[\left['\mathsf{cons}\, '\mathsf{var}\, (\pi_0\, E), \lambda i.\, ['\mathsf{k}\, (R\, i), (\pi_1\, E)\, i]\right], F\right] \end{array}$$

Just as in the universe of descriptions, this construction comes with an obvious *return* and a substitution operation, the *bind*. Its definition is the following:

$$\begin{array}{l} \mathsf{substI} : \ {}_{(I:\mathrm{SET})\to}(X, Y : I \to \mathrm{SET}) \to (R : \mathsf{TagIDesc}\, I) \\ \qquad (X \dot{\to} \mu^+_I\, (R^*_I\, Y)) \to \mu^+_I\, (R^*_I\, X) \dot{\to} \mu^+_I\, (R^*_I\, Y) \\ \mathsf{substI}\, X\, Y\, R\, \sigma\, i\, t \mapsto \\ \qquad \mathsf{catal}_I\, (\mathsf{de}\, R^*\, X)\, (\mu^+_Y\, (R^*\, Y))\, (\mathsf{applyI}\, R\, X\, Y\, \sigma)\, i\, t \end{array}$$

where applyI is defined as follows:

$$\begin{array}{l} \mathsf{applyI} : \ {}_{(I:\mathrm{SET})\to}(R : \mathsf{TagIDesc}\, I)(X, Y : I \to \mathrm{SET}) \to \\ \qquad (X \dot{\to} \mu^+_I\, (R^*_I\, Y)) \to \\ \qquad [\![\mathsf{de}\, R^*_I\, X]\!]_I\, \mu^+_I\, (R^*_I\, Y) \dot{\to} \mu^+_I\, (R^*_I\, Y) \\ \mathsf{applyI}\, R\, X\, Y\, \sigma\, i\, ['\mathsf{var}, x] \mapsto \sigma\, i\, x \\ \mathsf{applyI}\, R\, X\, Y\, \sigma\, i\, [c, ys] \ \mapsto \mathsf{con}\, [c, ys] \end{array}$$

The subscripted types corresponds to implicit arguments that can be automatically inferred, hence do not have to be typed in. Let us now consider two examples of free indexed monads.

Typed expressions: In the previous section, we presented a language of closed arithmetic expressions. Using the free monad construction, we are going to extend this construction to open terms. An open term is defined with respect to a context, represented by a snoc-list of types:

$$\begin{array}{ll} \mathsf{Context} : & \mathrm{SET} \\ [] & : \mathsf{Context} \\ \mathsf{snoc} & : \mathsf{Context} \to \mathsf{Ty} \to \mathsf{Context} \end{array}$$

An environment realises the context, packing a value for each type:

$$\begin{array}{l} \mathsf{Env} : \mathsf{Context} \to \mathrm{SET} \\ \mathsf{Env}\, [] \qquad\qquad \mapsto 1 \\ \mathsf{Env}\, (\mathsf{snoc}\, G\, S) \mapsto \mathsf{Env}\, G \times \mathsf{Val}\, S \end{array}$$

In this setting, we define type variables, Var by:

$$\begin{array}{ll} \mathsf{Var} \ : \mathsf{Context} \to \mathsf{Ty} \to \mathrm{SET} & \\ \mathsf{Var}\, [] & T \mapsto [] \\ \mathsf{Var}\, (\mathsf{snoc}\, G\, S) & T \mapsto (\mathsf{Var}\, G\, T) + (S == T) \end{array}$$

While Val maps the type to the corresponding host type, Var indexes a value in the context, obtaining a proof that the types match. The lookup function precisely follow this semantics:

$$\begin{array}{lll} \mathsf{lookup} : (G : \mathsf{Context}) \to \mathsf{Env}\, G \to (T : \mathsf{Ty}) \to \mathsf{Var}\, G\, T \to \mathsf{Val}\, T & & \\ \mathsf{lookup}\, (\mathsf{snoc}\, G\, T)\, [g, t]\, T\, (\mathsf{right}\ \mathsf{refl}) & \mapsto & t \\ \mathsf{lookup}\, (\mathsf{snoc}\, G\, S)\ [g, t]\, T\, (\mathsf{left}\ x) & \mapsto & \mathsf{lookup}\, G\, g\, T\, x \end{array}$$

Consequently, taking the free monad of ExprD by Var G, we obtain the language of open terms in a context G:

$$\mathsf{openTm}\, G \mapsto \mathsf{ExprD}^*_{\mathsf{Ty}}\, (\mathsf{Var}\, G)$$

In this setting, the language of closed terms corresponds to the free monad assigning an empty set of values to variables

$$\mathsf{closeTm} \mapsto \mathsf{ExprD}^*_{\mathsf{Ty}}\, \mathsf{Empty} \quad \text{where} \quad \begin{array}{l} \mathsf{Empty} : \mathsf{Ty} \to \mathrm{SET} \\ \mathsf{Empty}\, \_ \mapsto \#[] \end{array}$$

Allowing variables from an empty set is much like forbidding variables, so closeTm and ExprD describe isomorphic datatypes. Correspondingly, you can update an old ExprD to a shiny closeTm:

$$\begin{array}{l} \mathsf{update} : \mu^+_{\mathsf{Ty}}\, \mathsf{ExprD} \dot{\to} \mu^+_{\mathsf{Ty}}\, \mathsf{closeTm} \\ \mathsf{update}\, ty\, tm \mapsto \mathsf{catal}_{\mathsf{Ty}}\, (\mathsf{de}\, \mathsf{ExprD})\, (\mu^+_{\mathsf{Ty}}\, \mathsf{closeTm}) \\ \qquad\qquad (\lambda\_.\, \lambda[tag, tm].\, \mathsf{con}\, [1 + tag, tm])\, ty\, tm \end{array}$$

The other direction of the isomorphism is straightforward, the 'var case being impossible. Therefore, we are entitled to reuse the $\mathsf{eval}_{\Downarrow}$ function to define the semantics of closeTm.

Now we would like to give a semantics to the open term language. We proceed in two steps: first, we substitute variables by their value in the context; then, we evaluate the resulting closed term. Thanks to $\mathsf{eval}_{\Downarrow}$, the second problem is already solved. Let us focus on substituting variables from the context. Again, we can subdivide this problem: first, discharging a single variable from the context; then, applying this discharge function on every variables in the term.

The discharge function is relative to the required type and a context of the right type. Its action is to map values to themselves, and variables to their value in context. This corresponds to the following function:

$$\begin{array}{l} \mathsf{discharge} : \quad (G : \mathsf{Context}) \to \mathsf{Env}\, G \to \mathsf{Var}\, G \dot{\to} \mu^+_{\mathsf{Ty}}\, \mathsf{closeTm} \\ \mathsf{discharge}\, G\, g\, ty\, v \mapsto \mathsf{con}\, ['\mathsf{val}, \mathsf{lookup}\, G\, g\, ty\, v] \end{array}$$

We are now left with applying discharge over all variables of the term. We simply have to fill in the right arguments to substI, the type guiding us:

$$\begin{array}{l} \mathsf{substExpr} : \quad (G : \mathsf{Context}) \to \\ \qquad\qquad (\mathsf{Var}\, G \dot{\to} \mu^+_{\mathsf{Ty}}\, \mathsf{closeTm}) \dot{\to} \\ \qquad\qquad \mu^+_{\mathsf{Ty}}\, (\mathsf{openTm}\, G) \dot{\to} \mu^+_{\mathsf{Ty}}\, \mathsf{closeTm} \\ \mathsf{substExpr}\, G\, ty\, g\, \sigma\, tm \mapsto \mathsf{substI}_{\mathsf{Ty}}\, (\mathsf{Var}\, G)\, \mathsf{Empty}\, \mathsf{ExprD}\, \sigma\, ty\, tm \end{array}$$

Hence completing our implementation of the open terms interpreter. Without much effort, we have described the syntax of a well-typed language, together with its semantics.

Indexed descriptions: An interesting instance of free monad is IDesc itself. Indeed, 'var is nothing but the *return*. The remaining constructors form the carrier functor, trivially indexed by 1. The signature functor is described as follow:

$$\mathsf{IDescD}^{\mathsf{Sig}} : \mathsf{AlwaysD}\, 1$$

$$\mathsf{IDescD}^{\mathsf{Sig}} \mapsto \left[\begin{array}{l} ['\mathsf{k}\ \ '\times\ '\Sigma\ '\Pi], \\ \lambda\_.\, \left[\begin{array}{l} '\mathsf{k}\, \mathrm{SET} \\ '\mathsf{var}\, []\, '\times\, '\mathsf{var}\, [] \\ '\Sigma\, \mathrm{SET}\, (\lambda S.\, '\Pi\, S\, (\lambda\_.\, '\mathsf{var}\, [])) \\ '\Sigma\, \mathrm{SET}\, (\lambda S.\, '\Pi\, S\, (\lambda\_.\, '\mathsf{var}\, [])) \end{array}\right] \end{array}\right]$$

We get IDesc I by extending the signature with variables from I:

$$\begin{array}{l} \mathsf{IDescD} : (I : \mathrm{SET}) \to \mathsf{TagIDesc}\, 1 \\ \mathsf{IDescD}\, I \mapsto [\mathsf{IDescD}^{\mathsf{Sig}}, [\lambda\_.\, [], \lambda\_.\, []]]^*_1\, \lambda\_.\, I \end{array}$$

The fact that indexed descriptions are closed under substitution is potentially of considerable utility, if we can exploit this fact:

$$[\![\sigma D]\!]_J\, X \mapsto [\![D]\!]_I\, \lambda i.\, [\![\sigma i]\!]_J\, X \quad \text{where } \sigma : I \to \mathsf{IDesc}\, J$$

By observing that a description can be decomposed via substitution, we split its meaning into a superstructure of substructures, e.g. a 'database containing salaries', ready for traversal operations preserving the former and targeting the latter.

6. Discussion

In this paper, we have presented a universe of datatypes for a dependent type theory. We started from an unremarkable type theory with dependent functions and tuples, but relying on few other assumptions, especially where propositional equality is concerned. We added finite enumeration sufficient to account for constructor

choice, and then we built coding systems, first (as a learning experience) for simple ML-like inductive types, then for the indexed inductive families which dependently typed programmers in Agda, Coq and Epigram take for granted. We adopt a bidirectional type propagation mechanism to conceal artifacts of the encoding, giving a familiar and practicable constructor-based presentation to data.

Crucially to our approach, we ensure that the codes describing datatypes inhabit a datatype with a code. In a stratified setting, we avoid paradox by ensuring that this type of codes lives uniformly one level above the types the codes describe. The adoption of ordinary data to describe types admits datatype-generic operations implemented just by ordinary programming. In working this way, we make considerable use of type equality modulo open computation, silently specialising the types of generic operations as far as the datatype code for any given usage is known.

6.1 Related work in Generic Programming

Generic programming is a vast topic. We refer our reader to Garcia et al. [2003] for a broad overview of generic programming in various languages. For Haskell alone, there is a myriad of proposals: Hinze et al. [2007] and Rodriguez et al. [2008] provide useful comparative surveys.

Our approach follows the polytypic programming style, as initiated by PolyP [Jansson and Jeuring 1997]. Indeed, we build generic functions by induction on pattern functors, exploiting type-level computation to avoid the preprocessing phase: our datatypes are, natively, nothing but codes.

We have the *type-indexed datatypes* of Generic Haskell [Hinze et al. 2002] for free. From one datatype, we can compute others and equip them with relevant structure: the free monad construction provides one example. Our approach to encoding datatypes as data also sustains *generic views* [Holdermans et al. 2006], allowing us to rebias the presentation of datatypes conveniently. Tagged descriptions, giving us a sum-of-sigmas view, are a natural example.

Unlike Generic Haskell, we do not support polykinded programming [Hinze 2000]. Our descriptions are limited to endofunctors on SET^I. Whilst indexing is known to be sufficient to *encode* a large class of higher-kinded datatypes [Altenkirch and McBride 2002], we should rather hope to work in a more compositional style. We are free to write higher-order programs manipulating codes, but is not yet clear whether that is sufficient to deliver abstraction at higher kinds. Similarly, it will be interesting to see whether arity-generic programming [Weirich and Casinghino 2010] arises just by computing with our codes, or whether a richer abstraction is called for.

The Scrap Your Boilerplate [Lämmel and Peyton Jones 2003] (SYB) approach to generic programming offers a way to construct generic functions, based on dynamic type-testing via the Typeable type class. SYB cannot compute types from codes, but its dynamic character does allow a more flexible *ad hoc* approach to generic data traversal. By maintaining the correspondence between codes and types whilst supporting arbitrary inspection of codes, we pursue the same flexibility statically. The substitutive character of IDesc may allow us to observe and exploit *ad hoc* substructural relationships in data, but again, further work is needed if we are to make a proper comparison.

6.2 Generic Programming with Dependent Types

Generic programming is not new to dependent types. Altenkirch and McBride [2002] developed a universe of polykinded types in Lego; Norell [2002] gave a formalisation of polytypic programming in Alfa, a precursor to Agda; Verbruggen et al. [2008, 2009] provided a framework for polytypic programming in the Coq theorem prover. However, these works aim at *modelling* PolyP or Generic Haskell in a dependently-typed setting for the purpose of proving correctness properties of Haskell code. Our approach is different in that we aim at building a foundation for datatypes, in a dependently-typed system, for a dependently-typed system.

Closer to us is the work of Benke et al. [2003]. This seminal work introduced the usage of universes for developing generic programs. Our universes share similarities to theirs: our universe of descriptions is similar to their universe of iterated induction, and our universe of indexed descriptions is equivalent to their universe of finitary indexed induction. This is not surprising, as we share the same source of inspiration, namely induction-recursion.

However, we feel ready to offer a more radical prospectus. Their approach is generative: each universe extends the base type theory with both type formers and elimination rules. Thanks to levitation, we rely only on a generic induction and a specialised switchD, closing the type theory. We explore *programming* with codes, but also how to conceal the encoding when writing 'ordinary' programs.

6.3 Metatheoretical Status

The SET : SET approach we have taken in this paper is convenient from an experimental perspective, and it has allowed us to focus primarily on the encoding of universes, leaving the question of stratification (and with it, consistency, totality, and decidability of type checking) to one side. However, we must surely face up to the latter, especially since we have taken up the habit of constructing 'the set of all sets'. A proper account requires a concrete proposal for a system of stratified universes which allows us to make 'level-polymorphic' constructions, and we are actively pursuing such a proposal. We hope soon to have something to prove.

In the meantime, we can gain some confidence by systematically embedding predicative fragments of our theory within systems which already offer a universe hierarchy. We can, at the very least, confirm that in UTT-style theories with conventional inductive families of types [Luo 1994], as found in Coq (and in Agda if one avoids experimental extensions), we build the tower of universes we propose, cut off at an arbitrary height. It is correspondingly clear that some such system can be made to work, or else that other, longer-standing tools are troubled.

A metatheoretical issue open at time of writing concerns the size of the *index* set I in $\mathsf{IDesc}\,I$. Both Agda and recent versions of Coq allow inductive families with *large* indices, effectively allowing 'higher-kind' fixpoints on $\mathrm{SET}^{\mathrm{SET}}$ and more. They retain the safeguard that the types of *substructures* must be as small as the inductively defined superstructure. This liberalisation allows us large index sets in our models, but whilst it offers no obvious route to paradox by smuggling a large universe inside a small type, it is not yet known to be safe. We can restrict I as necessary to avoid paradox, provided 1, used to index IDesc itself, is 'small'.

6.4 Further Work

Apart from the need to nail down a stratified version of the system and its metatheory, we face plenty of further problems and opportunities. Although we have certainly covered Luo's criteria for inductive families [Luo 1994], there are several dimensions in which to consider expanding our universe.

Firstly, we seek to encompass *inductive-recursive* datatype families [Dybjer and Setzer 2001], allowing us to interleave the definition and interpretation of data in intricate and powerful ways. This interleaving seems particularly useful when reflecting the syntax of dependent type systems.

Secondly, we should very much like to extend our universe with a codes for internal fixpoints, as in [Morris et al. 2004]. The external knot-tying approach we have taken here makes types like 'trees with lists of subtrees' more trouble than they should be. Moreover, if we allow the alternation of least and greatest fixpoints,

we should expect to gain types which are not readily encoded with one external μ.

Thirdly, it would be fascinating to extend our universe with dedicated support for syntax with *binding*, not least because a universe with internal fixpoints has such a syntax. Harper and Licata have demonstrated the potential for and of such an encoding [Licata and Harper 2009], boldly encoding the invalid definitions along with the valid. A more conservative strategy might be to offer improved support for datatypes indexed by an extensible context of free variables, with the associated free monad structure avoiding capture as shown by Altenkirch and Reus [1999].

Lastly, we must ask how our new presentation of datatypes should affect the tools we use to build software. It is not enough to change the game: we must enable better play. If datatypes are data, what is design?

Acknowledgments

We are grateful to José Pedro Magalhães for his helpful comments on a draft of this paper. We are also grateful to the Agda team, without which levitation would have been a much more perilous exercise. J. Chapman was supported by the Estonian Centre of Excellence in Computer Science, EXCS, financed by the European Regional Development Fund. P.-É. Dagand, C. McBride and P. Morris are supported by the Engineering and Physical Sciences Research Council, Grants EP/G034699/1 and EP/G034109/1.

References

A. Abel, T. Coquand, and M. Pagano. A modular type-checking algorithm for type theory with singleton types and proof irrelevance. In *TLCA*.

R. Adams. Pure type systems with judgemental equality. *JFP*, 2006.

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2002.

T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*. 1999.

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *PLPV*, 2007.

L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter. Available at `http://www.cs.chalmers.se/~augustss/cayenne/interp.ps`, 1999.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 2003.

E. Brady, J. Chapman, P.-E. Dagand, A. Gundry, C. McBride, P. Morris, and U. Norell. An Epigram implementation.

E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *TYPES*, 2003.

J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

T. Coquand. An algorithm for type-checking dependent types. *SCP*, 1996.

J. Courant. Explicit universes for the calculus of constructions. In *TPHOLs*, 2002.

N. A. Danielsson. The Agda standard library, 2010.

P. Dybjer. Inductive sets and families in Martin-Löf's type theory. In *Logical Frameworks*. 1991.

P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In *TLCA*, 1999.

P. Dybjer and A. Setzer. Induction-recursion and initial algebras. In *Annals of Pure and Applied Logic*, 2000.

P. Dybjer and A. Setzer. Indexed induction-recursion. In *Proof Theory in Computer Science*. 2001.

R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA*, 2003.

H. Geuvers. Induction is not derivable in second order dependent type theory. In *TLCA*, 2001.

H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning and Computation*. 2006.

R. Harper and R. Pollack. Type checking with universes. In *TAPSOFT'89*.

R. Hinze. Polytypic values possess polykinded types. In *MPC*. 2000.

R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In *MPC*, 2002.

R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*. 2007.

S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In *MPC*. 2006.

P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *POPL*, 1997.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI*, 2003.

D. R. Licata and R. Harper. A universe of binding and computation. In *ICFP*, 2009.

Z. Luo. *Computation and Reasoning*. Oxford University Press, 1994.

P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

C. McBride and J. McKinna. The view from the left. *JFP*, 2004.

P. Morris. *Constructing Universes for Generic Programming*. PhD thesis, University of Nottingham, 2007.

P. Morris and T. Altenkirch. Indexed containers. In *LICS*, 2009.

P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. In *TYPES*, 2004.

P. Morris, T. Altenkirch, and N. Ghani. A universe of strictly positive families. *IJCS*, 2009.

U. Norell. Functional generic programming and type theory. Master's thesis, Chalmers University of Technology, 2002.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

N. Oury and W. Swierstra. The power of Pi. In *ICFP*, 2008.

C. Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur*. thèse d'habilitation, ENS Lyon, 1996.

B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, 1998.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell Symposium*, 2008.

The Coq Development Team. *The Coq Proof Assistant Reference Manual*.

W. Verbruggen, E. de Vries, and A. Hughes. Polytypic programming in Coq. In *WGP*, 2008.

W. Verbruggen, E. de Vries, and A. Hughes. Polytypic properties and proofs in Coq. In *WGP*, 2009.

S. Weirich and C. Casinghino. Arity-generic datatype-generic programming. In *PLPV*, 2010.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL*, 2003.

A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP*, 2009.