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. Surprisingly this apparently self-supporting setup is achievable without paradox or infinite regress.

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 2009], Epigram [McBride and McKinna 2004], and contemporary Haskell [Peyton Jones et al. 2006], 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], 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. 2009]. We have also developed a *stratified* model of our coding scheme in Agda[1].

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'

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

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

$$\begin{array}{lcl} \Gamma \vdash t : T & \Rightarrow & \Gamma \vdash \textsc{valid} \;\wedge\; \Gamma \vdash T : \textsc{Set} \\ \Gamma \vdash s \equiv t : T & \Rightarrow & \Gamma \vdash s : T \;\wedge\; \Gamma \vdash t : T \end{array}$$

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. 2009; 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 $\textsc{Set} : \textsc{Set}$, putting presentation before paradox [Girard 1972]. The usual remedies apply, *stratifying* $\textsc{Set}$ [Courant 2002; Harper and Pollack 1991; Luo 1994].

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

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

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

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

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

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

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

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.

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

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

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:

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

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:

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

This builds a right-nested tuple type, demanding an object of $P\, i$ for each i in the given finite domain. We can see these tuples as 'jump tables' tabulating dependently typed functions from the domain. We give this functional interpretation—the eliminator we need—by the switch operator, which, unsurprisingly, iterates projection:

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

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

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

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

$$\frac{\Gamma; x{:}S; \Delta \vdash \textsc{valid}}{\Gamma; x{:}S; \Delta \Vdash x \rhd x \in S} \qquad \frac{\begin{array}{c}\Gamma \Vdash f \rhd f' \in (x{:}S) \to T \\ \Gamma \Vdash S \ni s \rhd s'\end{array}}{\Gamma \Vdash f\ s \rhd f'\ s' \in T[s'/x]}$$

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

Figure 1. Type synthesis

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':

$$\begin{array}{c}\Gamma \Vdash e \rhd t \in T \Rightarrow \Gamma \vdash t : T \\ \Gamma \Vdash T \ni e \rhd t \Rightarrow \Gamma \vdash t : T\end{array}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2. Type checking

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 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: $\#[\text{'red 'blue}]$ is not $\#[\text{'green 'orange}]$.

3. A Universe of Inductive Datatypes

In this section, we describe an implementation of inductive types, as we know them in 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 of sums over arbitrary arities, where simply typed languages have finite sums.

On the other hand, our choice of notation $(x : A) \times (B x)$ emphasises that Σ-types generalise products, with the type of the second component depending on the value of the first, where 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 somehow for the recursive structure which ties these nodes together. Not for the first time, 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 seek to exploit 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 $D : \text{Desc}$ as a 'pattern functor' on SET, with $[\![D]\!]$ its action on an object, X, soon to be instantiated recursively.

Descriptions are sequential structures, terminated by '1, indicating the empty tuple. To build sigmas-of-sigmas, we define a $'\Sigma$ code, interpreted as a Σ-type. To request a recursive component, we have $\text{'ind}\times\ D$, where D describes the rest of the node.

You may have noticed that we are a little coy about this presentation, 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 $[\![D]\!]$. However, in Section 4, you will see what we actually do. In the meantime, let us first gain some intuition for its use by developing some examples.

$$\begin{array}{l}
\text{Desc} : \text{SET} \\
\text{'1} : \text{Desc} \\
'\Sigma\ (S : \text{SET})\ (D : S \to \text{Desc}) : \text{Desc} \\
\text{'ind}\times\ (D : \text{Desc}) : \text{Desc}
\end{array}$$

$$\begin{array}{llll}
[\![\_]\!] & : \text{Desc} \to \text{SET} \to \text{SET} & & \\
[\![\text{'1}]\!] & X & \mapsto & 1 \\
[\![{'\Sigma}\ S\ D]\!] & X & \mapsto & (s : S) \times [\![D\ s]\!]\ X \\
[\![\text{'ind}\times\ D]\!] & X & \mapsto & X \times [\![D]\!]\ X
\end{array}$$

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.

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

Let us explain its construction. First, we use $'\Sigma$ 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:

$$\text{NatD}\ Z \mapsto 1 + Z$$

Correspondingly, we can see the injections to the sum:

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

With a small change to this definition, we obtain the pattern functor for lists:

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

The 'suc constructor is turned into a proper 'cons, taking an argument in X followed by a recursive argument. This code describes the following functor:

$$\text{ListD}\ X\ Z \mapsto 1 + X \times Z$$

Finally, we are not limited to one recursive argument. This is demonstrated by our description of node-labelled binary trees:

$$\begin{array}{lll}
\text{TreeD} : \text{SET} \to \text{Desc} & & \\
\text{TreeD}\ X \mapsto {'\Sigma} & \#[\text{'leaf 'node}] \\
 & [\text{'1} \quad (\text{'ind}\times\ ({'\Sigma}\ X\ \lambda\_.\ \text{'ind}\times\ \text{'1}))]
\end{array}$$

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:

$$\text{TreeD}\ X\ Z \mapsto 1 + Z \times X \times Z$$

From the examples above, we observe that datatypes are defined by a $'\Sigma$ 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. Every description can be forced into this style with a singleton constructor if necessary. We characterise tagged descriptions thus:

$$\begin{array}{l}
\text{TagDesc} : \text{SET} \\
\text{TagDesc} \mapsto (E : \text{En}) \times (\pi\ E\ (\lambda\_.\ \text{Desc})) \\
\text{de} : \text{TagDesc} \to \text{Desc} \\
\text{de} \mapsto \wedge\lambda E.\ \lambda D.\ {'\Sigma}\ \#E\ (\text{switch}\ E\ (\lambda\_.\ \text{Desc})\ D)
\end{array}$$

It is not a great stretch to imagine that the traditional datatype declaration syntax might desugar to the *definition* of a datatype via 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:

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

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

$$\begin{array}{l} \mathsf{Nat} \mapsto \mu(\mathsf{de}\ [[\text{'}\mathsf{zero}\ \text{'}\mathsf{suc}], [\text{'}1 \quad (\text{'}\mathsf{ind}\!\times\ \text{'}1)]]) : \mathrm{SET} \\ \mathsf{con}\ [\text{'}\mathsf{zero}] : \mathsf{Nat} \qquad \mathsf{con}\ [\text{'}\mathsf{suc}\ (n : \mathsf{Nat})] : \mathsf{Nat} \end{array}$$

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':

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

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:

$$\begin{array}{l} \mathsf{ind} : (D : \mathsf{Desc})(P : \mu D \to \mathrm{SET}) \to \\ \qquad ((d : [\![D]\!]\ (\mu D)) \to \mathsf{All}\ D\ (\mu D)\ P\ d \to P(\mathsf{con}\ d)) \to \\ \qquad (x : \mu D) \to P x \\ \mathsf{ind}\ D\ P\ m\ (\mathsf{con}\ d) = m\ d\ (\mathsf{all}\ D\ (\mu D)\ P\ (\mathsf{ind}\ D\ P\ m)\ d) \end{array}$$

Here, $\mathsf{All}\ D\ X\ P\ d$ states that $P : X \to \mathrm{SET}$ 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 full definition (including an extra case, introduced shortly) is presented in Figure 4. Note that ind is our first generic operation over descriptions, albeit a hardwired operator. Any datatype we define automatically comes with an induction principle.

We note that the very same functors $[\![D]\!]$ also admit greatest fixpoints, and we have indeed implemented coinductive types this way, but that is a story for another time.

3.5 Extending type propagation

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

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

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.

$$\frac{\begin{array}{l}\Gamma \Vdash (d : [\![D]\!]\ (\mu D)) \to \mathsf{All}\ D\ (\mu D)\ (\lambda_{\mu D} x.\ P)\ d \to P[\mathsf{con}\ d/x] \\ \qquad \ni f \rhd f'\end{array}}{\Gamma \Vdash (x : \mu D) \to P \ni \circlearrowleft f \rhd \mathsf{ind}\ D\ (\lambda_{\mu D} x.\ P)\ f'}$$

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], but it does at least make 'assembly language' programming mercifully brief, if 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 universe. Persuading this 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 the reader 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 the En type former and its constructors. Recall:

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

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 discover that the type theory does not need to be extended with 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 $\mathsf{0}$ and $\mathsf{1{+}}$ 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

We shall now fulfil our implementation promises, encoding the universe of descriptions. In and of itself, the codes, Desc, is nothing but a datatype. We are in the same situation as with En: we ought to be able to describe the codes of Desc in Desc itself. Hence, this code would be a first-class citizen, born with the standard, generic equipment of datatypes.

4.2.1 First attempt

Our first attempt gets stuck quite quickly:

$$\begin{array}{l} \mathsf{DescD} : \mathsf{Desc} \\ \mathsf{DescD} \mapsto \mathsf{de} \left[\begin{bmatrix} \text{'}1 \\ \text{'}\Sigma \\ \text{'}\mathsf{ind}\!\times \end{bmatrix}, \begin{bmatrix} \text{'}1 \\ \text{'}\Sigma\ \mathrm{SET}\ (\lambda S.\ \{?\}) \\ \text{'}\mathsf{ind}\!\times\ \text{'}1 \end{bmatrix} \right] \end{array}$$

Let us explain where we stand. Much as we have done so far, we first offer a constructor choice from $\text{'}1$, $\text{'}\Sigma$, and $\text{'}\mathsf{ind}\!\times$. The reader will notice that the 'tagged' notation we have used for the Desc constructors now fully makes sense: these were actually the tags we are defining. For $\text{'}1$, we immediately reach the end of the description. For $\text{'}\mathsf{ind}\!\times$, there is a single recursive argument. Describing $\text{'}\Sigma$ is problematic. Recall the specification of $\text{'}\Sigma$:

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

So, we first pack a SET, S. We should then like a recursive argument *indexed* by S, but that is an *exponential*, and our presentation

$$\begin{array}{lllll}
\mathsf{All} : & \multicolumn{4}{l}{(D\!:\!\mathsf{Desc})(X\!:\!\textsc{Set})(P\!:\!X \to \textsc{Set})} \\
& \multicolumn{4}{l}{(xs\!:\![\![D]\!]\ X) \to \textsc{Set}} \\
\mathsf{All}\ \text{'1} & X\ P\ [] & = & 1 \\
\mathsf{All}\ (\text{'}\Sigma\ S\ D) & X\ P\ [s,d] & = & \mathsf{All}\ (D\ s)\ X\ P\ d \\
\mathsf{All}\ (\text{'ind}\times\ D) & X\ P\ [x,d] & = & P\ x \times \mathsf{All}\ D\ X\ P\ d \\
\mathsf{All}\ (\text{'hind}\times\ H\ D) & X\ P\ [f,d] & = & ((h\!:\!H) \to P\ (f\ h)) \times \mathsf{All}\ D\ X\ P\ d
\end{array}$$

$$\begin{array}{llll}
\mathsf{all} : & \multicolumn{3}{l}{(D\!:\!\mathsf{Desc})(X\!:\!\textsc{Set})(P\!:\!X \to \textsc{Set})} \\
& \multicolumn{3}{l}{(p\!:\!(x\!:\!X) \to P\ x)(xs\!:\![\![D]\!]\ X) \to \mathsf{All}\ D\ X\ P\ xs} \\
\mathsf{all}\ \text{'1} & X\ P\ p\ [] & = & [] \\
\mathsf{all}\ (\text{'}\Sigma\ S\ D) & X\ P\ p\ [s,d] & = & \mathsf{all}\ (D\ s)\ X\ P\ p\ d \\
\mathsf{all}\ (\text{'ind}\times\ D) & X\ P\ p\ [x,d] & = & [p\ x, \mathsf{all}\ D\ X\ P\ p\ d] \\
\mathsf{all}\ (\text{'hind}\times\ H\ D) & X\ P\ p\ [f,d] & = & [\lambda h.\, p\ (f\ h), \mathsf{all}\ D\ X\ P\ p\ d]
\end{array}$$

Figure 4. Defining and collecting inductive hypotheses

is entirely first-order so far, delivering 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. Intuitively, 'hind× gives a recursive subobject for each element of H.

$$\begin{array}{l}
\text{'hind}\times\ (H\!:\!\textsc{Set})\ (D\!:\!\mathsf{Desc}) : \textsc{Set} \\
[\![\text{'hind}\times\ H\ D]\!]\ X \mapsto (H \to X) \times [\![D]\!]\ X
\end{array}$$

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$(\lambda_.\, n)$. 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.[2]

We are now able to describe our universe of datatypes:

$$\begin{array}{l}
\mathsf{DescD} : \mathsf{Desc} \\
\mathsf{DescD} \mapsto \mathsf{de}\left[\left[\begin{array}{l}\text{'1}\\ \text{'}\Sigma\\ \text{'ind}\times\\ \text{'hind}\times\end{array}\right], \left[\begin{array}{l}\text{'1}\\ \text{'}\Sigma\ \textsc{Set}\ \lambda S.\ \text{'hind}\times\ S\ \text{'1}\\ \text{'ind}\times\ \text{'1}\\ \text{'}\Sigma\ \textsc{Set}\ \lambda\_.\ \text{'ind}\times\ \text{'1}\end{array}\right]\right]
\end{array}$$

The '1 and 'ind× cases remain unchanged, as expected. We successfully describe the 'Σ case, by a simple appeal to the higher-order induction on S. The 'hind× case consists in packing a Set with a recursive argument.

At a first glance, we have achieved our goal. We have described the codes of the universe of descriptions. Taking the fixpoint of $[\![\mathsf{DescD}]\!]$ gives us a datatype exactly like Desc. Might we be so bold as to take $\mathsf{Desc} \mapsto \mu\mathsf{DescD}$ 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 $\mathsf{Desc} \mapsto \mu\mathsf{DescD}$ is circular, but the offensive recursion is concealed by a prestidigitation. Expanding be de – and propagating types as in Figure 2 reveals the awful truth:

$$\begin{array}{l}
\mathsf{Desc} \mapsto \mu(\text{'}\Sigma\ \#[\text{'1}\ \text{'}\Sigma\ \text{'ind}\times\ \text{'hind}\times] \\
\qquad\qquad \mathsf{switch}\ [\text{'1}\ \text{'}\Sigma\ \text{'ind}\times\ \text{'hind}\times]\ (\lambda\_.\ \mathsf{Desc}) \\
\qquad\qquad \left[\begin{array}{l}\text{'1}\\ \text{'}\Sigma\ \textsc{Set}\ \lambda S.\ \text{'hind}\times\ S\ \text{'1}\\ \text{'ind}\times\ \text{'1}\\ \text{'}\Sigma\ \textsc{Set}\ \lambda\_.\ \text{'ind}\times\ \text{'1}\end{array}\right])
\end{array}$$

The recursion shows up only because we must specify the return type of the general-purpose switch, and it is computing a Desc! 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:

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

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.

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

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

We have levitated Desc. Beyond its pedagogical value, this exercise has several practical outcomes. First of all, it reveals that the Desc universe is just plain data. As any piece of data, it can therefore be inspected and manipulated. Moreover, it is expressed in the 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 now have all the necessary equipment in the theory to *program* over datatypes. *Generic programming is just programming.*

4.3 The generic catamorphism

In Section 3.4, we hardwired a dependent indunction principle, instead of the catamorphism. However, in some circumstances, the full power of a dependent elimination is not necessary. Let us now derive the catamorphism from ind principle.

The catamorphism is defined by induction on the description D, with a readily propagated non-dependent return type T. Given a node xs and the induction 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:

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

We are left with filling the hole. Recall that we have $xs : [\![D]\!]\ \mu D$ and $hs : \mathsf{All}\ D\ (\mu D)\ (\lambda\_.\ T)\ xs$ at hand. Our goal is to make an element of $[\![D]\!]\ T$. Intuitively, xs is of the right shape, but its sub-elements are of the wrong type. On the other hand, for each sub-element of xs, hs gives us the corresponding element in T. Therefore, to construct an element of $[\![D]\!]\ T$, we must replace the recursive components of xs by their counterparts from hs. Let us write a program to do that—please forgive us if we lapse to a

[2] E.g., extensionally, there is one inhabitant of $\#[] \to \mathsf{Nat}$; intensionally, there is a countable infinitude which it is not safe to collapse.

pattern matching notation, for readability's sake.

```
replace : (D : Desc)(X, Y : Set)
          (xs : ⟦D⟧ X) → All D X (λ_. Y) xs → ⟦D⟧ Y
replace '1 X Y [] [] ↦ []
replace ('Σ S D) X Y [s, d] d' ↦ [s, replace (D s) X Y d d']
replace ('ind× D) X Y [x, d] [y, d'] ↦
    [y, replace D X Y d d']
replace ('hind× H D) X Y [f, d'] [g, d'] ↦
    [g, replace D X Y d d']
```

Filling the hole in cata with replace D (μD) T xs hs closes the problem. In the type theory, we have built a generic catamorphism. Any datatype will now come equipped with this operation, for free.

With this example, we have shown how we can derive a generic operation, the catamorphism, from a pre-existing generic operation, the induction principle. This has been made possible by our ability to manipulate descriptions as first-class objects: the catamorphism is, basically, a function mapping a Desc to a datatype specific operation. This is a form of polytypic programming, as we learned from PolyP [Jansson and Jeuring 1997].

4.4 The generic free monad

In this section, we will turn to a more ambitious generic operation on datatype. Given a functor, represented as a tagged description, we build the free monad over this functor.

Let us recall the free monad construction. Given a functor F, the free monad over F is defined by the following datatype:

```
data FreeMonad (F : Set → Set)(X : Set) : Set where
  Var       : X → FreeMonad F X
  Composite : F(FreeMonad F X) → FreeMonad F X
```

Being an inductive type, this FreeMonad datatype is itself defined by a pattern functor. It is given by:

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

In our setting, the free monad construction will take the functor as a tagged description, a set X of variables, and will compute the tagged description of the corresponding free monad. Implementing this function is surprisingly easy:

```
_* : TagDesc → Set → TagDesc
[E, D]* X ↦ [['var , E], ['Σ X '1, D]]
```

We simply add a constructor, 'var, and define its argument to be a $'\Sigma\ X\ '1$, that is an element of X. We keep E and D as they were, hence leaving the other constructors unchanged. Unfolding the interpretation of this definition, we convince ourselves that this corresponds to the functor FreeMonadD. The fixpoint operation ties the knot and gives us the full-blown free monad construction.

Of course, we must equip the resulting datatypes with operations delivering a monadic interface. As expected, $\lambda x.\ '\mathsf{var}\ x$ plays the rôle of *return*, embedding variables into terms. The *bind* operation corresponds to *substitution*. We will now implement it, as a generic function.

Our implementation will appeal to the cata function developed previously. So, let us write down the types, and fill as much arguments to cata as possible:

```
subst :  (D : TagDesc)(X, Y : Set) → (X → μ(de (D* Y))) →
         μ(de (D* X)) → μ(de (D* Y))
subst D X Y σ ↦ cata (de (D* X)) (μ(de (D* Y))) {?}
```

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

```
apply :  (D : TagDesc)(X, Y : Set) → (X → μ(de D* X)) →
         ⟦de D* X⟧ μ(de D* Y) → μ(de D* Y)
apply D X Y σ ['var, x] ↦ σ x
apply D X Y σ [c, xs]   ↦ con [c, xs]
```

Filling the sub-goal with apply D X Y σ completes the implementation. To sum up, we have implemented the free monad construction for an arbitrary tagged description. This gives the developer the ability, for any datatype, to extend it with a notion of variable. Then, we have equipped this structure with the corresponding monadic operation, *bind* and *return*. This construction is an example of type-indexed datatype [Hinze et al. 2002], as found in Generic Haskell: from a datatype, we build a new datatype and equip it with its structure.

5. A Universe of Inductive Families

So far, we have explored the well-known realm of inductive types. We have built upon our intuition of ML-like datatypes. In our dependent setting, we have provided these datatypes by the mean of Desc, a universe of descriptions.

Working with dependent types fosters new opportunities for datatypes. The typical example is bounded lists, also known as vectors. A vector is a list decorated by its length. Having this information prevents hazardous operations, such as taking the head of an empty vector: the head function only takes vectors of length 'suc n, as enforced by its type. This is made possible by the specificity of dependent types: a term – the length – can influence a type – the vector type.

However, these datatypes cannot be defined by mere induction. In the case of vectors, for instance, we have to define the whole *family* of vectors in one go: vectors of all sizes need to be defined at the same time. In dependently-typed languages, the basic grammar of datatypes is that of inductive families. To capture this grammar, we rely on *indexing*.

5.1 The universe of indexed descriptions

In the previous section, we have presented the Desc universe as a grammar of functors in the category Set. We have seen how to code inductive types in this setting. To describe an inductive family indexed by I : Set, we use endofunctors on the category Set^I. We call these *indexed functors*. $I \to$ IDesc I is our grammar for describing these functors. Hence, IDesc and its interpretation have the following types:

```
IDesc (I : Set) : Set
⟦_⟧ :_{(I:Set)} → IDesc I → (I → Set) → Set
```

Given these components, we may interpret a *function* R : $I \to \mathsf{IDesc}\,I$ is interpreted as a function $I \to \mathrm{Set}^I \to \mathrm{Set}$, which is isomorphic to $\mathrm{Set}^I \to \mathrm{Set}^I$, the type of endofunctors on Set^I. Inductive families are fixpoints defined over these indexed functors, hence computing a fixpoint of the entire *family* of functors:

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

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

However, we still have to define the actual grammar. We obtain it by evolving Desc to cope with indexing. The code of IDesc is presented in Figure 6. Induction on indexed descriptions is defined

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

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

Figure 6. Universe of indexed descriptions

by:

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

Where the operators AllI and allI are presented in Figure 5. As for descriptions, we can compute a generic catamorphism, cataI, from indI.

5.2 Examples

Natural numbers: In order to gain some intuition of IDesc, let us re-implement the pattern functor of natural numbers:

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

Because Nat is just an inductive type, NatD is a 1-indexed functor. Therefore, the recursive argument is materialised by 'var [], where we were using 'ind× in the previous presentation. This transformation generalises to all inductive types. Moreover, we gain the ability to write mutually recursive inductive types.

Indexed descriptions: Note that IDesc I itself is merely an inductive type. Hence, we can describe it in IDesc 1:

$$
\begin{array}{l}
\mathsf{IDescD} : (I : \mathrm{Set}) \to \mathsf{IDesc}\ 1 \\
\mathsf{IDescD}\ I \mapsto \text{'}\Sigma \quad (\#[\text{'var 'const}\quad \text{'}{\times}\ \text{'}\Sigma\ \text{'}\Pi]) \\
\qquad\qquad\qquad \left[\begin{array}{l}
\text{'const}\ I \\
\text{'const}\ \mathrm{Set} \\
\text{'var}\ []\ \text{'}{\times}\ \text{'var}\ [] \\
\text{'}\Sigma\ \mathrm{Set}\ (\lambda S.\ \text{'}\Pi\ S\ (\lambda\_.\ \text{'var}\ [])) \\
\text{'}\Sigma\ \mathrm{Set}\ (\lambda S.\ \text{'}\Pi\ S\ (\lambda\_.\ \text{'var}\ []))
\end{array}\right]
\end{array}
$$

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

Vectors: So far, the examples we have seen live in IDesc 1, hence are not using any indexing. We remedy this by encoding the vectors. Recall that the constructors 'vnil and 'vcons are only defined for an index 'zero and 'suc respectively.:

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

One way to code constrained datatypes is to appeal to equality. The constraints are therefore captured by equations in the datatype. In this case, we obtain the following definition:

$$
\begin{array}{l}
\mathsf{VecD} : \mathrm{Set} \to \mathsf{Nat} \to \mathsf{IDesc}\ \mathsf{Nat} \\
\mathsf{VecD}\ X\ n = \text{'}\Sigma(\#[\text{'vnil 'vcons}]) \\
\qquad\qquad\qquad \left[\begin{array}{l}
\text{'const}\ (n == \text{'zero}) \\
\text{'}\Sigma\ \mathsf{Nat}\ \lambda m.\ \text{'var}\ m\ \text{'}{\times} \\
\qquad\qquad \text{'const}\ (n == \text{'suc}\ m)
\end{array}\right]
\end{array}
$$

In the 'vnil case, a proof must be provided that the index is equal to 'zero. In the 'vcons case, we first store an element m of Nat. However, the constraint stipulates that m cannot be *any* natural numbers: it must be "the index minus one". This translates into the constraint n == 'suc m, given a suitable presentation of propositional equality.

We have been careful to keep our setup agnostic with respect to notions of propositional equality. Any will do, according to your convictions, or for vectors, none—equality for Nat is definable by recursion—and many variations are popular. The traditional homogeneous identity type used in Coq is not adequate to support dependent pattern matching, but its heterogeneous variant, allowing equations between elements of arbitrary types, is sufficient to allow the translation of structurally recursive pattern matching programs to indI [Goguen et al. 2006]. Our present inclination is towards the extensional equality proposed by Altenkirch et al. [2007], which also sustains the translation.

However, sometimes, we can actually remove these equations altogether. Let us look back at Vec. We note that the equations are introduced because we are *storing* the index of the inductive family. However, *inductive families need not store their indices* [Brady et al. 2003]. By examining the incoming index, we can apply the *forcing* and *de-tagging* optimisations to our initial definition of Vec. This gives the following, equivalent definition:

$$
\begin{array}{lll}
\mathsf{VecD}\ (X : \mathrm{Set}) : \mathsf{Nat} \to \mathsf{IDesc}\ \mathsf{Nat} & & \\
\mathsf{VecD}\ X\ \text{'zero} & \mapsto & \text{'const}\ 1 \\
\mathsf{VecD}\ X\ (\text{'suc}\ n) & \mapsto & \text{'const}\ X\ \text{'}{\times}\ \text{'var}\ n
\end{array}
$$

The equations (and constructors) have simply disappeared. A similar example is Fin, 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 apply forcing, but not detagging, since both 'Fz and 'Fs both target 'suc:

$$
\begin{array}{lll}
\mathsf{FinD} : \mathsf{Nat} \to \mathsf{IDesc}\ \mathsf{Nat} & & \\
\mathsf{FinD}\ \text{'zero} & \mapsto & \text{'}\Sigma\ (\#[])\ [] \\
\mathsf{FinD}\ (\text{'suc}\ n) & \mapsto & \text{'}\Sigma(\#[\text{'Fz 'Fs}]) \\
 & & \quad \left[\begin{array}{l} \text{'const}\ 1 \\ \text{'var}\ n \end{array}\right]
\end{array}
$$

We should precise that forcing a description is not guaranteed to remove all constraints. It is subject to future work to see if constraints can be entirely eradicated, or presented more conveniently to the developer. Finally, it is worth mentioning that these optimisations are *source-to-source* transformations on descriptions.

Tagged indexed descriptions: When defining an indexed datatype, we have access to its index. Therefore, we can use this index to influence the choice of constructors. This captures the essence of dependent datatypes: a term – the index – has the ability to influence the datatype. We define tagged indexed descriptions to capture this specificity.

We divide a tagged indexed description in two parts: 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-depend part simply indexes the

$$
\begin{array}{llll}
\mathsf{AllI} : & \multicolumn{3}{l}{{}_{(I\,\textsc{Set})}\to (D\!:\!\mathsf{IDesc}\ I)(X : I \to \textsc{Set}) \to} \\
 & \multicolumn{3}{l}{[\![D]\!]_I\ X \to \mathsf{IDesc}\ ((i\!:\!I) \times X\ i)} \\
\mathsf{AllI}\ (\mathsf{'var}\ i) & X\ x & = & \mathsf{'var}\ [i, x] \\
\mathsf{AllI}\ (\mathsf{'const}\ K) & X\ k & = & \mathsf{'const}\ \mathsf{1} \\
\mathsf{AllI}\ (D\,\mathsf{'\!\times}\,D') & X\ [d, d'] & = & \mathsf{AllI}\,D\ X\ d\,\mathsf{'\!\times}\,\mathsf{AllI}\,D'\ X\ d' \\
\mathsf{AllI}\ (\mathsf{'}\Sigma\ S\ D) & X\ [s, d] & = & \mathsf{AllI}(D\ s)\ X\ d \\
\mathsf{AllI}\ (\mathsf{'}\Pi\ S\ D) & X\ f & = & \mathsf{'}\Pi\ S\ (\lambda s.\ \mathsf{AllI}(D\ s)\ X\ (f\ s))
\end{array}
$$

$$
\begin{array}{llll}
\mathsf{allI} : & \multicolumn{3}{l}{{}_{(I\,\textsc{Set})}\to (D\!:\!\mathsf{IDesc}\ I)(X : I \to \textsc{Set})(P\!:\!((i\!:\!I) \times X\ i) \to \textsc{Set}) \to} \\
 & \multicolumn{3}{l}{((x\!:\!(i\!:\!I) \times X\ i) \to P\ x) \to (xs\!:\![\![D]\!]_I\ X) \to [\![\mathsf{AllI}\,D\ X\ xs]\!]\ P} \\
\mathsf{allI}\ (\mathsf{'var}\ i) & X\ P\ p\ x & = & p\ [i, x] \\
\mathsf{allI}\ (\mathsf{'const}\ K) & X\ P\ p\ k & = & [] \\
\mathsf{allI}\ (D\,\mathsf{'\!\times}\,D') & X\ P\ p\ [d, d'] & = & [\mathsf{allI}\,D\ X\ P\ p\ d, \mathsf{allI}\,D'\ X\ P\ p\ d'] \\
\mathsf{allI}\ (\mathsf{'}\Sigma\ S\ D) & X\ P\ p\ [s, d] & = & \mathsf{allI}(D\ s)\ X\ P\ p\ d \\
\mathsf{allI}\ (\mathsf{'}\Pi\ S\ D) & X\ P\ p\ f & = & \mathsf{allI}(D\ a)\ X\ P\ p\ (f\ a)
\end{array}
$$

Figure 5. Indexed induction predicates

choice of constructors by I. Hence, by inspecting the index, it is possible to enable or disable constructors.

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

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

We use the notation $\overline{TID}$ to denote the indexed description computed from the tagged indexed description TID. Its expansion is similar to the definition of de but more involved.

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

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

An expression of this language is either a value, a conditional expression, an addition of numbers, or a comparison of numbers. Informally, their type is the following:

$$
\begin{array}{lll}
\mathsf{'cond} & : & \forall ty\!:\!\mathsf{Ty}.\mathsf{'bool} \to ty \to ty \to ty \\
\mathsf{'plus} & : & \mathsf{'nat} \to \mathsf{'nat} \to \mathsf{'nat} \\
\mathsf{'le} & : & \mathsf{'nat} \to \mathsf{'nat} \to \mathsf{'bool} \\
\mathsf{'val} & : & \forall ty\!:\!\mathsf{Ty}.\mathsf{Val}\ ty \to ty
\end{array}
$$

The function Val, used in the definition of 'val, simply maps a type ty of the object language to the corresponding type in the host language. Hence, the arguments of 'val are ensured to be of the expected type. We assume that Nat and Bool represent natural numbers and booleans in the host language, equipped with an addition operation plusHost and a comparison function leHost. We define Val as follows:

$$
\begin{array}{lll}
\mathsf{Val} & : & \mathsf{Ty} \to \textsc{Set} \\
\mathsf{Val}\ \mathsf{'nat} & = & \mathsf{Nat} \\
\mathsf{Val}\ \mathsf{'bool} & = & \mathsf{Bool}
\end{array}
$$

In our universe of descriptions, the syntax of this language is described by a tagged indexed description. We use the index to carry the type: the resulting description is indexed by Ty. We observe that some constructors are always defined, namely 'cond and 'val. On the other hand, the 'plus and 'le constructors are index-dependent. 'plus is defined if and only if the result type – the index – is 'nat, whereas 'le is defined if and only if the index is 'bool. The actual code precisely follows this intuition, as shown in Figure 7.

Having implemented the syntax, we would like to describe its semantics. To do so, we implement an evaluator. The type of the evaluator is:

$$\mathsf{eval}_{\Downarrow} : (ty\!:\!\mathsf{Ty}) \to \mu_{\mathsf{Ty}}\overline{\mathsf{ExprD}}\ ty \to \mathsf{Val}\ ty$$

The type of $\mathsf{eval}_{\Downarrow}$ is strikingly similar to a catamorphism. Indeed, implementing a single step of evaluation – the algebra – is sufficient, as catal gives, for free, the full evaluator. The implementation is as follows:

$$
\begin{array}{l}
\mathsf{ExprD} : \mathsf{TagIDesc}\ \mathsf{Ty} \\
\mathsf{ExprD} \mapsto (\mathsf{ExprAD}, \mathsf{ExprID})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{ExprAD} : \mathsf{AlwaysD}\ \mathsf{Ty} \\
\mathsf{ExprAD} \mapsto \left[\begin{array}{l}
[\mathsf{'val}\ \mathsf{'cond}], \\
\lambda ty. \left[\begin{array}{l}
\mathsf{'const}\ (\mathsf{Val}\ ty) \\
\mathsf{'var}\ \mathsf{'bool}\,\mathsf{'\!\times}\,\mathsf{'var}\ ty\,\mathsf{'\!\times}\,\mathsf{'var}\ ty
\end{array}\right]
\end{array}\right]
\end{array}
$$

$$
\begin{array}{l}
\mathsf{ExprID} : \mathsf{IndexedD}\ \mathsf{Ty} \\
\mathsf{ExprID} \mapsto \left[\begin{array}{l}
[[\mathsf{'plus}]\quad[\mathsf{'le}]], \\
\lambda\_.\ \mathsf{'var}\ \mathsf{'nat}\,\mathsf{'\!\times}\,\mathsf{'var}\ \mathsf{'nat}
\end{array}\right]
\end{array}
$$

Figure 7. Syntax of typed expressions

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathsf{eval}_{\downarrow} : (ty\!:\!\mathsf{Ty}) \to [\![\overline{\mathsf{ExprD}}\ ty]\!]_{\mathsf{Ty}}\ \mathsf{Val} \to \mathsf{Val}\ ty} \\
\mathsf{eval}_{\downarrow}\ \_ & [\mathsf{'val}, x] & = & x \\
\mathsf{eval}_{\downarrow}\ \_ & [\mathsf{'cond}, [\mathsf{'true}, [x, \_]]] & = & x \\
\mathsf{eval}_{\downarrow}\ \_ & [\mathsf{'cond}, [\mathsf{'false}, [\_, y]]] & = & y \\
\mathsf{eval}_{\downarrow}\ \mathsf{'nat} & [\mathsf{'plus}, [x, y]] & = & \mathsf{plusHost}\ x\ y \\
\mathsf{eval}_{\downarrow}\ \mathsf{'bool} & [\mathsf{'le}, [x, y]] & = & \mathsf{leHost}\ x\ y
\end{array}
$$

$$
\begin{array}{l}
\mathsf{eval}_{\Downarrow} : (ty\!:\!\mathsf{Ty}) \to \mu_{\mathsf{Ty}}\overline{\mathsf{ExprD}}\ ty \to \mathsf{Val}\ ty \\
\mathsf{eval}_{\Downarrow}\ ty\ term = \mathsf{catal}_{\mathsf{Ty}}\overline{\mathsf{ExprD}}\ \mathsf{Val}\ \mathsf{eval}_{\downarrow}\ ty\ term
\end{array}
$$

Hence, we have defined the syntax of a typed language of arithmetic and boolean expressions. We have given its semantics through an evaluation function. Provided a one step semantic of the language, the big step interpreter is granted without effort thanks to the generic catamorphism.

However, so far, we are only able to define and manipulate *closed* terms. By abstracting over Val, it is possible to build and manipulate *open* terms, that is, terms with variables. Following Val, we define Var by:

$$
\begin{array}{lll}
\mathsf{Var} & : & \mathsf{En} \to \mathsf{Ty} \to \textsc{Set} \\
\mathsf{Var}_{dom}\ \_ & = & \#dom
\end{array}
$$

Whereas Val was mapping the type to the corresponding host type, Var maps types to a finite set. The finite set – the context – contains closed terms. A variable is therefore a 'val that contains a pointer to a particular element of the finite set – an element of $\#dom$.

Consequently, replacing $\mathsf{Val}\ ty$ by $(\mathsf{Val}\ ty{+}\mathsf{Var}_{dom}\,ty)$ in Figure 7 turns the language of closed terms into a language of opened terms with variables and constants. For readability, we abbreviate $\lambda ty.\,\mathsf{Val}\ ty{+}\mathsf{Var}_{dom}\,ty$ by $\mathsf{Val}{+}\mathsf{Var}_{dom}$. This defines a new indexed description, called $\mathsf{ExprD}_{\mathsf{Var},dom}$.

Again, we would like to give a semantics to this extended language. We proceed in two steps: first, we replace the 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 discharging 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 themself, and variables to their value in context. This corresponds to the following function:

$$\begin{array}{ll}
\mathsf{discharge} : & (ty : \mathsf{Ty})(dom : \mathsf{En}) \\
 & (\gamma : \pi\ dom\ (\lambda\_.\ \mu_{\mathsf{Ty}} \overline{\mathsf{ExprD}_{\mathsf{Var},dom}}\ ty)) \to \\
 & (\mathsf{Val}\ ty + \mathsf{Var}_{dom}\ ty) \to \mu_{\mathsf{Ty}} \overline{\mathsf{ExprD}_{\mathsf{Var},dom}}\ ty \\
\multicolumn{2}{l}{\mathsf{discharge}\ ty\ dom\ \gamma\ (\mathsf{left}\ x) \mapsto \mathsf{con}\ [\text{'}\mathsf{val}, x]} \\
\multicolumn{2}{l}{\mathsf{discharge}\ ty\ dom\ \gamma\ (\mathsf{right}\ v) \mapsto} \\
\multicolumn{2}{l}{\quad \mathsf{switch}\ dom\ (\lambda\_.\ \mu_{\mathsf{Ty}} \overline{\mathsf{ExprD}_{\mathsf{Var},dom}}\ ty)\ \gamma\ v}
\end{array}$$

We are now left with applying discharge over all variables of the term. The type of this operation is the following:

$$\begin{array}{ll}
\mathsf{substExpr} : & (dom : \mathsf{En}) \\
 & (\gamma_{\mathsf{nat}} : \Gamma_{\mathsf{ty}}\ dom\ \text{'}\mathsf{nat})(\gamma_{\mathsf{bool}} : \Gamma_{\mathsf{ty}}\ dom\ \text{'}\mathsf{bool}) \\
 & (\sigma : \quad (dom : \mathsf{En}) \\
 & \qquad (\gamma_{\mathsf{nat}} : \Gamma_{\mathsf{ty}}\ dom\ \text{'}\mathsf{nat}) \\
 & \qquad (\gamma_{\mathsf{bool}} : \Gamma_{\mathsf{ty}}\ dom\ \text{'}\mathsf{bool}) \\
 & \qquad (ty : \mathsf{Ty}) \to (\mathsf{Val}\ ty + \mathsf{Var}_{dom}\ ty) \to \\
 & \qquad \mu_{\mathsf{Ty}} \overline{\mathsf{ExprD}}\ ty) \to \\
 & (ty : \mathsf{Ty}) \to \mu_{\mathsf{Ty}} \overline{\mathsf{ExprD}_{\mathsf{Var},dom}}\ ty \to \\
 & \mu_{\mathsf{Ty}} \overline{\mathsf{ExprD}}\ ty
\end{array}$$

Where Γ_{ty} corresponds to a context, defined by:

$$\begin{array}{l}
\Gamma_{\mathsf{ty}} : \mathsf{En} \to \mathsf{Ty} \to \mathrm{SET} \\
\Gamma_{\mathsf{ty}}\ dom\ ty = \pi\ dom\ (\lambda\_.\ \mu_{\mathsf{Ty}} \overline{\mathsf{ExprD}_{\mathsf{Var},dom}}\ ty)
\end{array}$$

Abstracting away the book-keeping introduced by contexts, this definition looks familiar. It is similar to a monadic *bind*. This is not surprising as we are defining a first-order syntax with variables: our datatype enjoys more structure than what we are given. We are facing a free monad, where 'val is the *return* introducing variables. For convenience, we wrap discharge in a σ function that picks the context of the right type:

$$\sigma\ dom\ \gamma_{\mathsf{nat}}\ \gamma_{\mathsf{bool}}\ ty\ var \mapsto \mathsf{discharge}\ ty\ dom\ \gamma_{ty}\ var$$

Where γ_{ty} is short for **case** ty **of** $\left\{ \begin{array}{l} \text{'}\mathsf{nat} \to \gamma_{\mathsf{nat}} \\ \text{'}\mathsf{bool} \to \gamma_{\mathsf{bool}} \end{array} \right.$

Instead of implementing substExpr in this special case, we are now going to implement 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:

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

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:

$$\begin{array}{ll}
\mathsf{substI} : & {}_{(I:\mathrm{SET})} \to (R : \mathsf{TagIDesc}\ I)(X, Y : I \to \mathrm{SET}) \to \\
 & ((i : I) \to X\ i \to \mu_I (\overline{R^{*}_{I}\ Y})\ i) \to \\
 & (i : I)(D : \mu_I (\overline{R^{*}_{I}\ X})\ i) \to \mu_I (\overline{R^{*}_{I}\ Y})\ i \\
\multicolumn{2}{l}{\mathsf{substI}\ X\ Y\ R\ \sigma\ i\ t =} \\
\multicolumn{2}{l}{\quad \mathsf{cataI}\ \overline{R^{*}\ X}\ (\mu \overline{R^{*}\ Y})\ (\mathsf{applyI}\ R\ X\ Y\ \sigma)\ i\ t}
\end{array}$$

Where applyI is defined as follow:

$$\begin{array}{ll}
\mathsf{applyI} : & {}_{(I:\mathrm{SET})} \to (R : \mathsf{TagIDesc}\ I)(X, Y : I \to \mathrm{SET}) \to \\
 & ((i : I) \to X\ i \to \mu_I \overline{(R^{*}_{I}\ Y)\ i}) \to \\
 & (i : I) \to [\![\overline{(R^{*}_{I}\ X)\ i}]\!]_I\ \mu_I \overline{(R^{*}_{I}\ Y)} \to \mu_I \overline{(R^{*}_{I}\ Y)\ i} \\
\multicolumn{2}{l}{\mathsf{applyI}\ R\ X\ Y\ \sigma\ i\ [\text{'}\mathsf{var}, x] \mapsto \sigma\ i\ x} \\
\multicolumn{2}{l}{\mathsf{applyI}\ R\ X\ Y\ \sigma\ i\ [c, ys] \mapsto \mathsf{con}\ [c, ys]}
\end{array}$$

Let us now consider two examples of free indexed monad.

Typed expressions: In Section 5.2, we had the intuition that our datatypes ExprD and $\mathsf{ExprD}_{\mathsf{Var},dom}$ enjoy a monadic structure. We had identified the variable substitution operation as the *bind* of a free monad. To exhibit its monadic structure, we first have to massage the definition of our datatype.

As previously mentioned, we identify 'val with the *return* of the free monad, while the other components are the action of the monad. As a result, the definition is similar to ExprD presented in Figure 7, replacing ExprAD by $\mathsf{ExprAD}^{\mathsf{Free}}$:

$$\begin{array}{l}
\mathsf{ExprAD}^{\mathsf{Free}} : \mathsf{AlwaysD}\ \mathsf{Ty} \\
\mathsf{ExprAD}^{\mathsf{Free}} \mapsto \left[\begin{array}{l} \#[\text{'}\mathsf{cond}], \\ \lambda ty.\ [\ \text{'}\mathsf{var}\ \text{'}\mathsf{bool}\ \text{'}{\times}\ \text{'}\mathsf{var}\ ty\ \text{'}{\times}\ \text{'}\mathsf{var}\ ty\ ] \end{array} \right]
\end{array}$$

We call this datatype $\mathsf{ExprD}^{\mathsf{Free}}$. By a simple unfolding of definition, we note that $\mathsf{ExprD}^{\mathsf{Free}}{}^{*}_{\mathsf{Ty}}\ \mathsf{Val}$ corresponds to the syntax of closed terms, ExprD. Similarly, $\mathsf{ExprD}^{\mathsf{Free}}{}^{*}_{\mathsf{Ty}}\ (\mathsf{Val}{+}\mathsf{Var}_{dom})$ corresponds to expressions with variables, $\mathsf{ExprD}_{\mathsf{Var},dom}$.

The evaluator for closed terms we implemented in Section 5.2 remains unchanged. It reduces closed terms in $\mathsf{ExprD}^{\mathsf{Free}}{}^{*}_{\mathsf{Ty}}\ \mathsf{Val}\ ty$ to values in $\mathsf{Val}\ ty$. We are left with implementing substExpr. We simply have to fill in the right arguments to substI, the type guiding us:

$$\begin{array}{l}
\mathsf{substExpr}\ dom\ \gamma_{\mathsf{nat}}\ \gamma_{\mathsf{bool}}\ \sigma\ ty\ term \mapsto \\
\quad \mathsf{substI}_{\mathsf{Ty}} \quad \mathsf{ExprD}^{\mathsf{Free}}\ (\mathsf{Val}{+}\mathsf{Var}_{dom})\ \mathsf{Val} \\
\qquad\qquad\quad (\sigma\ dom\ \gamma_{\mathsf{nat}}\ \gamma_{\mathsf{bool}})\ ty\ term
\end{array}$$

Hence completing our implementation of the open terms interpreter.

We have defined a well-typed language of arithmetical expressions, taking advantage of indexing. Then, we have implemented an evaluator for closed term, based on the generic catamorphism function. Using the free monad construction, we have automatically derived the language of open terms. Using its monadic structure, we have implemented the interpreter for open terms in context. Hence, without much efforts, we have described the syntax of a well-typed language, together with its semantics.

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

$$\begin{array}{l}
\mathsf{IDescD}^{\mathsf{Free}} : \mathsf{AlwaysD}\ 1 \\
\mathsf{IDescD}^{\mathsf{Free}} \mapsto \left[\begin{array}{l} [\text{'}\mathsf{const}\ \ \text{'}{\times}\ \text{'}\Sigma\ \text{'}\Pi], \\ \lambda\_.\ \left[\begin{array}{l} \text{'}\mathsf{const}\ \mathrm{SET} \\ \text{'}\mathsf{var}\ []\ \text{'}{\times}\ \text{'}\mathsf{var}\ [] \\ \text{'}\Sigma\ \mathrm{SET}\ (\lambda S.\ \text{'}\Pi\ S\ (\lambda\_.\ \text{'}\mathsf{var}\ [])) \\ \text{'}\Sigma\ \mathrm{SET}\ (\lambda S.\ \text{'}\Pi\ S\ (\lambda\_.\ \text{'}\mathsf{var}\ [])) \end{array} \right] \end{array} \right]
\end{array}$$

Then, we get IDesc by building its free monad:

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

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

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

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.

In this section, we have presented the universe of indexed description. It embraces indexed families of types and, as such, allows us to write dependent datatypes. Hence, we have presented several example of indexed datatypes. In this context, we have presented the free monad construction, together with its monadic operations.

6. Discussion

6.1 Universe stratification

As such, our type theory suffers from an inconsistency. Indeed, the typing rule $\mathrm{Set} : \mathrm{Set}$ leads to Girard's paradox. We made that choice for presentational convenience, as universe stratification is orthogonal to our work. Nonetheless, our universe of description stratifies naturally. IDesc is self-encoding only in a level-polymorphic sense. Unsurprisingly, IDesc at level l is of type Set^{l+1}. Similarly, the interpretation of IDesc at level l is an object of type Set^{l}:

$$
\begin{array}{ll}
\mathsf{IDesc}^{l}(I:\mathrm{Set}^{l+1}) & :\mathrm{Set}^{l+1} \\
\text{'var}\ (i:I) & :\mathsf{IDesc}^{l}\,I \\
\text{'const}\ (A:\mathrm{Set}^{l}) & :\mathsf{IDesc}^{l}\,I \\
(D:\mathsf{IDesc}^{l}\,I)\ \text{'}{\times}\ (D:\mathsf{IDesc}^{l}\,I) & :\mathsf{IDesc}^{l}\,I \\
\text{'}\Sigma\ (S:\mathrm{Set}^{l})\ (D:S\to\mathsf{IDesc}^{l}\,I) & :\mathsf{IDesc}^{l}\,I \\
\text{'}\Pi\ (S:\mathrm{Set}^{l})\ (D:S\to\mathsf{IDesc}^{l}\,I) & :\mathsf{IDesc}^{l}\,I \\
\multicolumn{2}{l}{[\![\_]\!]^{l} :_{(I:\mathrm{Set}^{l+1})\to}\ \mathsf{IDesc}^{l}\ I \to (I\to\mathrm{Set}^{l})\to\mathrm{Set}^{l}} \\
\ldots &
\end{array}
$$

Crucially, the types of data stored in an $\mathsf{IDesc}^{l}\,I$ all live no higher—we may store an I and a Set^{l} in a Set^{l+1}. The code for $\mathsf{IDesc}^{l}\,I$ is an element of $\mathsf{IDesc}^{l+1}1$, so there is a spiral, not a cycle. We have checked the construction using Agda's universe polymorphism, coding IDesc in itself and have proving the isomorphism between the host and the embedded universes.

6.2 Related work

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. In the sole context of Haskell, there is a myriad of proposals. These approaches are compared in Hinze et al. [2007] and Rodriguez et al. [2008].

Our approach is follow the polytypic programming style, as initiated by PolyP [Jansson and Jeuring 1997]. Indeed, we build generic functions by induction on pattern functors. Unlike PolyP, we do not have to resort to preprocessing: our datatypes are, natively, nothing but codes.

We share with Generic Haskell the *type-indexed datatype* approach [Hinze et al. 2002], as exemplified by the free monad construction: from datatype, we can compute new datatypes and equip them with their structure. Generic Haskell also features *generic views* [Holdermans et al. 2006], transparently transforming the structure of datatype definitions. An example is the tagged descriptions, presenting datatypes under a sum-of-sigmas angle. Unlike Generic Haskell, we do not have to modify the compiler to obtain views on datatypes: we can massage descriptions from inside our language.

Unlike Generic Haskell, we do not support polykinded programming [Hinze 2000]. Our descriptions are limited to endofunctors on Set and Set^{I}. While we could *encode* higher-kinded datatypes, we do not plan to adopt this strategy. As future work, we plan to extend our universe to capture higher-kinded definitions and generic functions over them. For the same reason, arity-generic programming [Weirich and Casinghino 2010] is out of reach of our current presentation.

Another generic programming paradigm is Scrap Your Boilerplate [Lämmel and Peyton Jones 2003] (SYB). Our proposal is different in various ways. The corner stone of SYB is the *spine* view of datatype constructors. A piece of data is a spine composed by a constructor applied to some arguments. SYB provides a combinator library to write generic functions over spines. This relies on a Typeable type-class, allowing dynamic dispatch to datatype-specific operations. As a result, SYB is not reflexive: it is restricted to datatypes instanciating Typeable. Moreover, it is limited to building generic functions, hence type-indexed datatypes cannot be implemented in this framework.

Generic programming in dependent types is not new either. Norell [2002] has given a formalization of polytypic programming in Alfa, a precursor of Agda. Similarly, Verbruggen et al. [2008, 2009] have developed 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 differ in several ways. First, there approach is generative: each universe extends the base type theory with both type formers and elimination rules. Thanks to levitation, we only rely on a generic induction and a specialised switchD. Second, the authors do not tackle the issue of *programming* with codes. We have shown how to abstract away codes and give a convenient presentation to the developer. The authors often resort to an extensional equality, while we have given an equality-agnostic presentation. Beside, our universes are arranged so as to use definitional equality as much as possible. Hence, in practice, the developer is relieved from many proof obligations.

7. Conclusion

In this paper, we have presented a universe of datatypes for a dependent type theory. To ensure the generality of our proposal, this system has been built in a familiar type theory, with no assumption about the underlying propositional equality. Because our approach is extensively using codes for universes, we have given a rationalised presentation of codes. Thanks to type propagation, we make practical the usage of codes for datatypes.

To introduce our approach, we have presented a universe of description. This universe has the expressive power of simple inductive types, as found in ML-like languages. Further, we have implemented this universe as a self-described object. Hence, for a minimal extension of the type-theory, we get a closed, self-describing presentation of datatypes, where datatypes are just data.

To capture dependent datatypes, we generalise our presentation to support indexing. The universe of indexed descriptions thus built encompasses inductive families. Again, this universe is self-described. We have developed several examples of dependent datatypes and generic functions over them.

We have presented a self-describing, self-hosted universe for datatypes. We have shown the benefit of such approach, by our ability to reflect datatypes in the type-theory. This fosters a new way of considering generic programming: just as programming.

Moreover, despite its egg-and-chicken nature, this presentation is free of paradox: it has been formalised in Agda, admitting a correct stratification.

Future work: As such, indexed descriptions do not cover several extensions of inductive families. One of them is induction-recursion. An interesting question is to locate indexed descriptions in the spectrum between inductive families and indexed induction-recursion. Another popular extension we plan to consider is to allow internal fixpoints and higher-kinded datatypes.

Also, we have presented a generic notion of syntax with variables, thanks to the free monad construction. We would like to explore a notion of syntax with binding. Interestingly, introducing internal fixpoints or kinds would turn our universe into such syntax with binding. Once again, levitation would reveal itself convenient by providing generic tools to handle binding.

Acknowledgments

We are gratefully to José Pedro Magalhães for his helpful comments on a draft of this paper. 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 P.-L. Curien, editor, *TLCA*, volume 5608 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2009. ISBN 978-3-642-02272-2.

R. Adams. Pure type systems with judgemental equality. *J. Funct. Program.*, 16(2):219–246, 2006.

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *PLPV '07*, pages 57–68. ACM, 2007.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4): 265–289, 2003.

E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003. ISBN 3-540-22164-6.

E. Brady, J. Chapman, P.-E. Dagand, A. Gundry, C. McBride, P. Morris, and U. Norell. An Epigram implementation, 2009. URL `http://www.e-pig.org/darcs/Pig09/src/Epitome.pdf`.

T. Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996.

J. Courant. Explicit universes for the calculus of constructions. In *TPHOLs '02*, pages 115–130, London, UK, 2002.

N. A. Danielsson. The Agda standard library.

P. Dybjer. Inductive sets and families in Martin-Löf's type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. CUP, 1991.

P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *TLCA*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999. ISBN 3-540-65763-0.

P. Dybjer and A. Setzer. Induction-recursion and initial algebras. In *Annals of Pure and Applied Logic*, volume 124, 2000.

R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA '2003*, pages 115–134, 2003.

H. Geuvers. Induction is not derivable in second order dependent type theory. In *TLCA*, pages 166–181, 2001.

J.-Y. Girard. *Interprétation functionelle et Elimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.

H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning and Computation*, Lecture Notes in Computer Science, chapter 27, pages 521–540. 2006.

R. Harper and R. Pollack. Type checking with universes. In *TAPSOFT '89*, pages 107–136, 1991.

R. Hinze. Polytypic values possess polykinded types. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, chapter 2, pages 2–27. 2000.

R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In *MPC '02*, pages 148–174, 2002.

R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, chapter 2, pages 72–149. 2007.

S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In T. Uustalu, editor, *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 209–234. 2006.

P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *POPL '97*, pages 470–482, 1997.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In Z. Shao and P. Lee, editors, *TLDI*, pages 26–37. ACM, 2003. ISBN 1-58113-649-8.

Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, USA, May 1994.

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

C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, January 2004.

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

P. Morris and T. Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009.

P. Morris, T. Altenkirch, and N. Ghani. A universe of strictly positive families. *International Journal of Foundations of Computer Science*, 20(1):83–107, 2009.

U. Norell. Functional generic programming and type theory. Master's thesis, Computing Science, Chalmers University of Technology, 2002. Available from `http://www.cs.chalmers.se/~ulfn`.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

N. Oury and W. Swierstra. The power of Pi. In J. Hook and P. Thiemann, editors, *ICFP*, pages 39–50. ACM, 2008. ISBN 978-1-59593-919-7.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06*, pages 50–61, New York, NY, USA, 2006. ACM.

B. C. Pierce and D. N. Turner. Local type inference. In *POPL'98*, pages 252–265, 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 A. Gill, editor, *Haskell Symposium*, pages 111–122. ACM, 2008. ISBN 978-1-60558-064-7.

The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.2*, 2009.

W. Verbruggen, E. de Vries, and A. Hughes. Polytypic programming in coq. In *WGP '08*, pages 49–60, 2008.

W. Verbruggen, E. de Vries, and A. Hughes. Polytypic properties and proofs in coq. In *WGP '09*, pages 1–12, 2009.

S. Weirich and C. Casinghino. Arity-generic datatype-generic programming. In *PLPV '10*, pages 15–26, 2010.