A Foundation for GADTs and Inductive Families

Dependent Polynomial Functor Approach

Makoto Hamana
Gunma University
hamana@cs.gunma-u.ac.jp

Marcelo Fiore
University of Cambridge
Marcelo.Fiore@cl.cam.ac.uk

Abstract

Every Algebraic Datatype (ADT) is characterised as the initial algebra of a polynomial functor on sets. This paper extends the characterisation to the case of more advanced datatypes: Generalised Algebraic Datatypes (GADTs) and Inductive Families. Specifically, we show that GADTs and Inductive Families are characterised as initial algebras of dependent polynomial functors. The theoretical tool we use throughout is an abstract notion of polynomial between sets together with its associated general form of polynomial functor between categories of indexed sets introduced by Gambino and Hyland.

In the context of ADTs, this fundamental result is the basis for various generic functional programming techniques. To establish the usefulness of our approach for such developments in the broader context of inductively defined dependent types, we apply the theory to construct zippers for Inductive Families.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data types and Structures; F.3.3 [*Logics and Meaning of Programs*]: Studies of Program Constructs—Type structure

General Terms Theory, Languages

Keywords Dependent types, categorical semantics

1. Introduction

It is well-known that every Algebraic Datatype (ADT) is characterised as the initial algebra of a polynomial functor, see *e.g.* (Hagino 1987). This representation of ADTs is the basis for various generic functional programming techniques, such as:

- fold and fusion techniques (Meijer et al. 1991; Sheard and Fegaras 1993; Launchbury and Sheard 1995; Takano and Meijer 1995; Hu et al. 1996; Ghani et al. 2005; Katsumata and Nishimura 2008; Hinze 2010),
- polytypic programming (Jansson and Jeuring 1997),
- Generic Haskell (Hinze and Jeuring 2003),
- program reasoning (Danielsson et al. 2006), and
- generic zippers (McBride 2001; Morihata et al. 2009).

WGP'11, September 18, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0861-8/11/09...$10.00

The scene of functional languages is however rapidly shifting to incorporate programming with some kind of dependent types. Hence, the generic programming techniques listed above should be extended also to this setting. To tackle the problem in a principled manner one needs first to consider algebraic foundations for inductively defined dependent types, exhibiting them as initial algebras of a class of polynomial functors. These can then be used to reformulate and extend the known generic programming techniques for ADTs.

This paper is concerned with inductively defined dependent types as given by Generalised Algebraic Datatypes (GADTs) and Inductive Families (IFs). We show that they are characterised as initial algebras of dependent polynomial functors between indexed sets (as induced by an abstract notion of polynomial between sets) introduced by (Gambino and Hyland 2003). This fundamental result provides links between ADTs, GADTs, and IFs. Thus enabling the transfer of technologies developed in one world (*e.g.* ADTs) to another one (*e.g.* GADTs) using the following relationships amongst these classes of datatypes:

$$\begin{tikzcd} \text{ADTs} \arrow[r, "\text{extend to}"] & \text{GADTs} \arrow[r, bend left, "\text{extend to}"] & \text{Inductive Families} \arrow[l, bend left, "\text{mimicked by}"] \end{tikzcd}$$

As an application of the theory realising the above, we give a construction of zipper data structures for IFs that extends the one for ADTs.

Contributions. The contributions of the paper are:

- elaboration on the polynomial representation of various GADTs and IFs that automatically generate polynomial functors for them;
- presentation of a simple proof of the existence and construction of initial algebras of polynomial functors for GADTs and IFs;
- development of zippers for IFs.

Our work aims to establish foundations for *generic programming* with dependent types as understood in the following two senses:

- *datatype generic*, in that programs that may be instantiated to different types are not redefined for each one;
- *type-theory generic*, in that programming principles apply universally (for any language that supports dependent types, such as Haskell, Agda (Norell 2007), Coq, Epigram (McBride 2004), Marin-Löf Type Theory, and the Calculus of Inductive Constructions (CIC)).

The foundations of ADTs by means of initial algebras of polynomial functors have achieved these two goals. For instance, principles based on this representation hold in any language that supports ADTs.

This paper achieves the goal of datatype genericity. As for the second goal of type-theory genericity, we restrict attention to *total* type theories and programming. Thus targeting the total fragments of Haskell and Agda.

The development of type-theory generic foundations requires an abstract mathematical approach, not relying on syntactic formulations. Thus, encoding formalisms within some particular powerful type theory (as *e.g.* CIC) does not match our purposes. Hence, we take a category-theoretic approach. We are however not concerned here in providing the most general categorical framework covering all known languages and type theories. Instead, we work with a concrete mathematical universe appropriate for the total type theory and functional programming settings, *viz.* the category of sets.

Organisation. The paper is organised as follows. We recall the notions of polynomial between sets and of polynomial functors between categories of indexed sets in Section 2. We characterise GADTs and IFs, by means of polynomials, as initial algebras for polynomials functors in Sections 3 and 4. In Section 5, we apply the theory to the construction of zipper data structures for IFs. Finally, in Section 6, we discuss related work.

2. Polynomials

Indexed sets. The universes of discourse for modelling GADTs and IFs considered in this paper are categories of indexed sets.

For a set I, the category $\mathbf{Set}^I$ has

- *objects:* $A : I \to \mathbf{Set}$, *i.e.* I-indexed sets $\{A(i) \mid i \in I\}$;
- *arrows:* I-indexed functions $f : A \to B$, *i.e.* functions $f(i) : A(i) \to B(i)$ for all $i \in I$.

We note that this construction is parametric in the indexing set, and we will indeed need to use it in a variety of cases $\mathbf{Set}^I$, $\mathbf{Set}^J$, $\mathbf{Set}^{I\times J}$, $\mathbf{Set}^U$, *etc.*

We need recall three important functors between categories of indexed sets. These arise from a function $h : I \to J$ as follows:

$$\begin{aligned} h^* &: \mathbf{Set}^J \to \mathbf{Set}^I & h^*(A) &\overset{def}{=} A \circ h \ , \\ \Sigma_h &: \mathbf{Set}^I \to \mathbf{Set}^J & \Sigma_h(A)(j) &\overset{def}{=} \sum_{\substack{i\in I\\ j\equiv h(i)}} A(i) \ , \\ \Pi_h &: \mathbf{Set}^I \to \mathbf{Set}^J & \Pi_h(A)(j) &\overset{def}{=} \prod_{\substack{i\in I\\ j\equiv h(i)}} A(i) \ . \end{aligned}$$

We call the functor h^* *reindexing*, the functor Σ_h *dependent sum*, and the functor Π_h *dependent product*. Note that in the definition of the latter two the index i is subject to the condition $j \equiv h(i)$, where "$\equiv$" denotes the equality of elements in a set. Note also that a functor $F : \mathbf{Set}^I \to \mathbf{Set}^J$ can be regarded as "curried", so that it takes two arguments $X \in \mathbf{Set}^I$ and $j \in J$ to give $F(X)(j) \in \mathbf{Set}$.

Polynomials. We consider an abstract form of polynomials and polynomial functors due to (Gambino and Hyland 2003). These mathematical structures will be used to express GADTs and IFs.

Definition 2.1 A *(dependent) polynomial* P is a triple $P = (d, p, c)$ of functions between sets as follows:

$$\begin{tikzcd} I & E \arrow[l, "d"'] \arrow[r, "p"] & B \arrow[r, "c"] & J \end{tikzcd} \ .$$

Given a polynomial as above, respectively applying the reindexing, dependent product, and dependent sum to each of its functions, one obtains the series of functors

$$\begin{tikzcd} \mathbf{Set}^I \arrow[r, "d^*"] & \mathbf{Set}^E \arrow[r, "\Pi_p"] & \mathbf{Set}^B \arrow[r, "\Sigma_c"] & \mathbf{Set}^J \end{tikzcd} \ . \qquad (1)$$

Note that reindexing reverses the direction of the arrow.

Definition 2.2 The *(dependent) polynomial functor F_P* associated to a (dependent) polynomial $P = (d, p, c)$ is defined as the composite (1); that is,

$$F_P(X) \overset{def}{=} \Sigma_c(\Pi_p(d^*(X))).$$

We typically omit the subscript P when the polynomial is clear from the context.

A functor between categories of indexed sets is said to be polynomial whenever it is isomorphic to one induced by a polynomial.

We strictly distinguish between the intensional descriptions provided by polynomials and the extensional counterparts that they generate in the form of polynomial functors. This can be understood as in ordinary mathematics, where one distinguishes the syntactic expression x^2 and the mathematical function that squares its argument.

Polynomial functors on **Set** are instances of dependent polynomial functors for $I = J = 1$. On the other hand, by expanding Def. 2.2, dependent polynomial functors can be described concretely as systems of polynomial functors in many variables. Indeed, the description

Polynomial functor of a polynomial $P = (d, p, c)$:

$$F_P(X)(j) \ = \ \sum_{\substack{b\in B\\ j\equiv c(b)}} \ \prod_{\substack{e\in E\\ b\equiv p(e)}} X(d(e))$$

exhibits the dependent polynomial functor $F_P : \mathbf{Set}^I \to \mathbf{Set}^J$ as a "sum of products" functor. More precisely, it is a J-indexed system of polynomial functors in I variables

$$\{F_P(X)(j) \ \mid \ j \in J\}$$

with $F_P(X)(j)$ given by the sum over b in $B_j = \{b \in B \mid c(b) \equiv j\}$ of the monomials consisting of the product of the $X(d(e))$ for e ranging over $E_b = \{e \in E \mid p(e) \equiv b\}$.

Polynomials have several interesting closure properties, see *e.g.* (Gambino and Hyland 2003; Kock 2009; Gambino and Kock 2010). Indeed, they are closed under sums, products, composition, parameterised initial algebras, and differentiation.

In this paper, we use the fundamental result that every polynomial endofunctor has an initial algebra (see Thms. 3.5 and 4.1, and Props. 3.9 and 4.2) and the operations of sum and differentiation. The sum of a family of polynomials $\{P_k = (d_k, p_k, c_k) \mid k \in K\}$, for K an arbitrary set, is the polynomial

$$\Sigma_{k\in K} P_k = \begin{tikzcd} I & \Sigma_{k\in K} E_k \arrow[l, "{[d_k]_{k\in K}}"'] \arrow[r, "\Sigma_{k\in K} p_k"] & \Sigma_{k\in K} B_k \arrow[r, "{[c_k]_{k\in K}}"] & J \end{tikzcd}$$

for which $F_{\Sigma_{k\in K} P_k} \cong \Sigma_{k\in K} F_{P_k}$. The operation of differentiation is discussed in §5.

3. GADTs as Polynomials

We begin with the general definition of GADTs.

We let α, ε range over type variables, and τ, σ range over *mono types* as defined by the grammar

$$\tau ::= \alpha \mid b \mid \tau \to \tau' \mid D\,\overline{\tau}$$

where b is a base type and D is a type constructor. We use the notation $\overline{\alpha}^n$ for a sequence $\alpha_1, \ldots, \alpha_n$ of length n (the superscript n may be omitted). Hereafter, we only treat mono types, and just call them types.

A GADT D is a type constructor specified in the following Haskell-like syntactic form (Schrijvers et al. 2009).

$$\begin{array}{l} \texttt{data}\ D : *^n \to * \ \texttt{where} \\ \quad K : \forall \overline{\alpha}^l, \overline{\varepsilon}^m.\, \tau_1 \to \cdots \to \tau_k \to D\,\overline{\sigma} \end{array}$$

Here $*^n \to *$ denotes $* \to \cdots \to * \to *$ (with $\to$ iterated n-times), K is a constructor, the type $\overline{\sigma}$ can only contain type variables from $\overline{\alpha}$, and each type τ_i can contain both type variables from $\overline{\alpha}$ and $\overline{\varepsilon}$. The type variables $\overline{\varepsilon}$ can be seen as existentially quantified, as they do not appear in the conclusion of the constructor (*i.e.* the right-hand side of the final $\to$).

Notation 3.1 Throughout the paper we will adopt the above format to present datatype declaration schema. While the scheme contains only one constructor, we implicitly assume that the declaration provides finitely many of them. This simplifies the presentation.

An example of GADT is the GADT `Fin` of "bounded natural numbers", see *e.g.* (Johann and Ghani 2008), defined by in Haskell notation as

```
data Z; data S a

data Fin :: * -> * where
  Zero :: Fin (S a)
  Succ :: Fin a -> Fin (S a)
```

Here `Z` and `S a` are distinct *types* (not data), defined to be empty. Universal quantification is omitted in Haskell's GADT declarations. The example does not involve existential type variables. The type `Fin` n types natural numbers below n, *e.g.*

`Succ Zero :: Fin (S (S Z)).`

In particular, the type `Fin Z` is not inhabited.

In the following subsections, we proceed to give initial-algebra semantics for GADTs by means of polynomials. We do this in a stepwise fashion, considering in turn *Simple GADTs*, *Simple GADTs with constant types*, and *Positive GADTs*. We assume that there are no mutual definitions of GADTs in a program. Thus, any datatype appearing in the declaration of a GADT D must be constructed without using D.

3.1 Simple GADTs

We consider first the following restricted form of GADTs, referred to as *Simple GADTs*.

Simple GADT

$$\begin{array}{l} \texttt{data}\ D : *^n \to * \ \texttt{where} \\ \quad K : \forall \overline{\alpha}^l, \overline{\varepsilon}^m.\, D(d_1[\overline{\alpha},\overline{\varepsilon}]) \to \cdots \to D(d_k[\overline{\alpha},\overline{\varepsilon}]) \to D(c[\overline{\alpha}]) \end{array}$$

Assumption: D does not appear in every d_i and c.

The notation $e[\overline{\alpha}]$ stands for an expression e that may contain type variables in $\overline{\alpha}$. The d_i and c are sequences of types of length n.

Simple GADTs provide the most basic form of GDTs that depart from ADTs. `Fin` is an example of a Simple GADT, where $c = \texttt{S}$, and there is no $\overline{\varepsilon}$.

To model a Simple GADT declaration as a polynomial

$$\begin{tikzcd} I & E \arrow[l, "d"'] \arrow[r, "p"] & B \arrow[r, "c"] & J \end{tikzcd}$$

we need first to define the sets I, E, B, J. In general, elements of these sets are *indices* of types. What are the indices of a GADT D? They are themselves *types*. Hence we need to collect all types used for defining the GADT D. The set of *all closed types* U is thus defined as

$$U \ni \tau \ ::= \ b \mid \tau \to \tau' \mid D\,\overline{\tau}.$$

Modelling `Fin`. We start by showing how to model the GADT `Fin` as a prototypical example of a Simple GADT. We then generalise the analysis to arbitrary Simple GADTs.

First, we describe each constructor as a polynomial. We denote by ! the unique function from the empty set $\varnothing$ to a set. The constructor `Zero :: Fin (S a)` is modelled by the polynomial

$$Zero = \qquad \begin{tikzcd} U & \varnothing \arrow[l, "!"'] \arrow[r, "!"] & U \arrow[r, "\mathtt{S}"] & U. \end{tikzcd}$$

The polynomial for `Succ :: Fin a -> Fin (S a)` is

$$Succ = \qquad \begin{tikzcd} U & U \arrow[l, "\mathrm{id}"'] \arrow[r, "\mathrm{id}"] & U \arrow[r, "\mathtt{S}"] & U. \end{tikzcd}$$

These definitions are systematically generated according to the constructor specification. For instance, for

`Succ :: Fin a -> Fin (S a),`

the domain `Fin a` corresponds to the first id in the polynomial *Succ*, and the codomain `Fin (S a)` corresponds to the `S` in the polynomial. The middle id is due to `Succ` being unary.

Applying the construction (1) to each polynomial we obtain their corresponding polynomial functors.

$$\begin{array}{ll} F_{Zero}: & \mathbf{Set}^U \xrightarrow{!^*} \mathbf{Set}^{\varnothing} \xrightarrow{\Pi_!} \mathbf{Set}^U \xrightarrow{\Sigma_{\mathtt{S}}} \mathbf{Set}^U \\ F_{Succ}: & \mathbf{Set}^U \xrightarrow{\mathrm{id}^*} \mathbf{Set}^U \xrightarrow{\Pi_{\mathrm{id}}} \mathbf{Set}^U \xrightarrow{\Sigma_{\mathtt{S}}} \mathbf{Set}^U \end{array}$$

The polynomial functor $F_{Fin} : \mathbf{Set}^U \to \mathbf{Set}^U$ for the GADT `Fin` is defined as the sum

$$\begin{aligned} F_{Fin}(X)(\tau) &\overset{def}{=} F_{Zero}(X)(\tau) + F_{Succ}(X)(\tau) \\ &= \Sigma_{\mathtt{S}}\Pi_!!^*(X)(\tau) + \Sigma_{\mathtt{S}}\Pi_{\mathtt{id}}\mathrm{id}^*(X)(\tau) \\ &= \sum_{\substack{a\in U \\ \tau\equiv \mathtt{S}\,a}} \prod_{e\in\varnothing} X(!(e)) + \sum_{\substack{a\in U \\ \tau\equiv \mathtt{S}\,a}} \prod_{\substack{e\in U \\ a\equiv \mathrm{id}(e)}} X(\mathrm{id}(e)) \\ &= \sum_{\substack{a\in U \\ \tau\equiv \mathtt{S}\,a}} 1 + \sum_{\substack{a\in U \\ \tau\equiv \mathtt{S}\,a}} X(a) \quad = \sum_{\substack{a\in U \\ \tau\equiv \mathtt{S}\,a}} 1 + X(a) \end{aligned}$$

The definition of this functor is equivalent to the definition by pattern-matching

$$\begin{aligned} F_{Fin}(X)(\mathtt{S}\,a) &= 1 + X(a) \\ F_{Fin}(X)(a) &= \varnothing \qquad\qquad \text{otherwise.} \end{aligned}$$

It is a general phenomenon that the equality constraint under which dependent sums are subject expresses pattern-matching. This is also known as the Henry Ford encoding (see the discussion in §6).

Crucially, one can ensure that the functor F_{Fin} has an initial algebra by a general construction (see Thm. 3.5). This can be calculated as the union of the ω-chain

$$\mathbf{0} \subseteq F_{Fin}(\mathbf{0}) \subseteq F^2_{Fin}(\mathbf{0}) \subseteq \cdots$$

where $\mathbf{0} = \{\varnothing \mid u \in U\} \in \mathbf{Set}^U$. For $n > 0$, the ω-chain at an index $\mathtt{S}^n\, a$ is

$$\varnothing \ \subseteq \ \{\mathtt{Zero}\} \ \subseteq \ \{\mathtt{Zero}\} + \{\mathtt{Succ}\ a \mid a \in \{\mathtt{Zero}\}\} \ \subseteq \ \cdots$$

The union $\mathsf{Fin} \in \mathbf{Set}^U$ of these, for $a \in U$, is thus described as

$$\mathsf{Fin}(\mathtt{S}^n\ a) = \{\mathtt{Zero}, \mathtt{Succ\ Zero}, \cdots, \mathtt{Succ}^{n-1}\ \mathtt{Zero}\},$$
$$\mathsf{Fin}(a) = \varnothing \qquad \text{otherwise.}$$

This certainly models the GADT `Fin`.

General case. Let D be a Simple GADT with constructors $K_1, \ldots, K_l$. Since $d_i[\overline{\alpha}, \overline{\varepsilon}]$ is a sequence of types of length n, seen as taking types $\overline{\alpha}, \overline{\varepsilon}$ as inputs, it is modelled as a function $d_i : U^{l+m} \to U^n$. Similarly, $c[\overline{\alpha}]$ is modelled as a function $c : U^l \to U^n$.

Notation 3.2 Given two functions $f : A \to C, g : B \to C$, we write $[f,g] : A+B \to C$ for the function $[f,g](x) = f(x)$ if $x \in A$, $[f,g](x) = g(x)$ if $x \in B$. We also use the function $\nabla = [\mathrm{id}_A, \mathrm{id}_A] : A+A \to A$, and more generally $\nabla_k : kA \to A$ as its k-folded version (here kA denotes the k-fold sum $A + \cdots + A$).

The polynomial P_i for a constructor K_i of a Simple GADT is defined as

$$\boxed{\begin{tikzcd} U^n & kU^{l+m} \arrow[l, "{[d_1,\ldots,d_k]}"'] \arrow[r, "\nabla_k"] & U^{l+m} \arrow[r, "c\pi_l"] & U^n \end{tikzcd}}$$

where $\pi_l : U^{l+m} \to U^l$ is the projection: $\pi_l(\overline{\tau}^l, \tau_{l+1}, \ldots, \tau_{l+m}) = \overline{\tau}^l$. This induces the polynomial functor $F_{P_i} : \mathbf{Set}^{U^n} \to \mathbf{Set}^{U^n}$ given by

$$F_{P_i}X(\overline{\tau}) = \sum_{\substack{\overline{\alpha}\in U^l, \overline{\varepsilon}\in U^m \\ \overline{\tau}\equiv c(\overline{\alpha})}} X(d_1(\overline{\alpha},\overline{\varepsilon})) \times \cdots \times X(d_k(\overline{\alpha},\overline{\varepsilon}))$$

Notation 3.3 As above, for ease of reference, we hereafter present the polynomial of a datatype within double-framed boxes and indicate its corresponding polynomial functor with a left vertical bar.

The polynomial functor $F_P : \mathbf{Set}^{U^n} \to \mathbf{Set}^{U^n}$ for the Simple GADT D is the sum of the above:

$$F_P X(\overline{\tau}) = F_{P_1}X(\overline{\tau}) + \cdots + F_{P_l}X(\overline{\tau}).$$

(Recall that as noted in §2 the sum of polynomial functors is again a polynomial functor.)

We record the following corollary of Thm. 3.5.

Proposition 3.4 *The polynomial functor of a Simple GADT has an initial algebra.*

3.2 Simple GADTs with constant types

Simple GADTs lack constant parameters within their definition. Consider the GADT `Vec` of length-indexed lists, called vectors.

```
data Vec :: * -> * where
  Nil  :: Vec Z
  Cons :: Int -> Vec n -> Vec (S n)
```

This declaration involves the constant type `Int`, not covered by the previous account. So we extend the class of Simple GADTs as follows.

Simple GADT with constant types

$\mathtt{data}\ D : *^n \to *\ \mathtt{where}$

$K : \forall \overline{\alpha}^l, \overline{\varepsilon}^m.\ Q[\overline{\alpha},\overline{\varepsilon}] \to D(d_1[\overline{\alpha},\overline{\varepsilon}]) \to \cdots \to D(d_k[\overline{\alpha},\overline{\varepsilon}])$
$\qquad \to D(c[\overline{\alpha}])$

Here Q introduces "constant types" as a sequence of arbitrary types that may contain type variables, D does not appear in Q, and c is a sequence of types of length n.

Comprehension of an indexed set. We present a construction that collects all elements of an indexed set. This is needed to define the polynomial for GADTs with constants.

Let $A \in \mathbf{Set}^I$. The disjoint union of the family $\{A(i) \mid i \in I\}$ is usually defined by the union of index-labelled sets as

$$\begin{aligned} \uplus_{i\in I} A(i) &\overset{def}{=} \cup_{i\in I} \{i\} \times A(i) \\ &= \{(i,a) \mid i \in I,\ a \in A(i)\}. \end{aligned}$$

Hence the disjoint union operation constructs a single set from an indexed set. We use the new notation $\boldsymbol{\{}A\boldsymbol{\}}$ for $\uplus_{i\in I} A_i$, and call it *the comprehension of* A, as it comprehensively collects all elements of A. For a pair $(i,a) \in \boldsymbol{\{}A\boldsymbol{\}}$, the first component is an index $i \in I$, and the second an element $a \in A(i)$. We define the "projection" function π that selects the corresponding index of an element by

$$\pi : \boldsymbol{\{}A\boldsymbol{\}} \to I; \quad \pi(i,a) = i.$$

(The theoretical background for the notion of comprehension is discussed in §6.5).

Polynomial for a constant. Using comprehension, one can define a constant type $Q \in \mathbf{Set}^U$ as the polynomial P_Q

$$\begin{tikzcd} U & \varnothing \arrow[l, "!"'] \arrow[r, "!"] & \boldsymbol{\{}Q\boldsymbol{\}} \arrow[r, "\pi"] & U \end{tikzcd}.$$

Indeed, calculating the polynomial functor $F_{P_Q} : \mathbf{Set}^U \to \mathbf{Set}^U$, we have

$$\begin{aligned} F_{P_Q}X(\tau) &= \Sigma_\pi \Pi_! !^*(X)(\tau) \\ &= \sum_{\substack{(i,t)\in\boldsymbol{\{}Q\boldsymbol{\}} \\ \tau\equiv\pi(i,t)}} 1 = \sum_{(\tau,t)\in\boldsymbol{\{}Q\boldsymbol{\}}} 1 = \sum_{t\in Q(\tau)} 1 \cong Q(\tau). \end{aligned}$$

The important points in this derivation are: $(\tau,t) \in \boldsymbol{\{}Q\boldsymbol{\}} \Leftrightarrow t \in Q(\tau)$ (by definition) and $\sum_{s\in S} 1 \cong S$ (for every set S).

Interpretation of a type. We also prepare the way to interpret types. The polynomial of a GADT D is denoted P_D, and we use the shorter notation F_D for its associated polynomial functor F_{P_D}. The initial F_D-algebra is denoted μF_D. Each base type b is associated with some set B; *e.g.* `Int` is associated with $\mathbb{Z}$.

The interpretation $[\![\overline{\alpha}^l \vdash \tau]\!]$ of a mono type τ under the type variable context $\overline{\alpha}$ is an indexed set in $\mathbf{Set}^{U^l}$, taking closed types $\overline{\sigma} \in U^l$ to give the set $[\![\overline{\alpha}^l \vdash \tau]\!](\overline{\sigma})$ interpreting τ instantiated at $\overline{\sigma}$. The definition is as follows:

$$\begin{aligned} [\![\overline{\alpha} \vdash \alpha_i]\!](\overline{\sigma}) &= [\![\vdash \sigma_i]\!] \\ [\![\overline{\alpha} \vdash b]\!](\overline{\sigma}) &= B \\ [\![\overline{\alpha} \vdash \tau \to \tau']\!](\overline{\sigma}) &= [\![\overline{\alpha} \vdash \tau]\!](\overline{\sigma}) \to [\![\overline{\alpha} \vdash \tau']\!](\overline{\sigma}) \\ [\![\overline{\alpha} \vdash D\,\overline{\tau}^n]\!](\overline{\sigma}) &= (\mu F_D)([\![\overline{\alpha} \vdash \tau_1]\!](\overline{\sigma}), \cdots, [\![\overline{\alpha} \vdash \tau_n]\!](\overline{\sigma})) \end{aligned}$$

We remark that, even though a polynomial P_D may use the interpretation function, the definition is not circular because we do not consider mutual GADTs in this paper.

Polynomial for a Simple GADT with constant types. We define the polynomial for Simple GADTs with constants.

Suppose the "constant type" part $Q[\overline{\alpha},\overline{\varepsilon}]$ in a declaration of a Simple GADT with constant types to be a sequence $Q_1[\overline{\alpha},\overline{\varepsilon}], \ldots, Q_p[\overline{\alpha},\overline{\varepsilon}]$ of types. Each type $Q_i[\overline{\alpha},\overline{\varepsilon}]$ is modelled as the indexed set $\widetilde{Q}_i = [\![\overline{\alpha},\overline{\varepsilon} \vdash Q_i]\!] \in \mathbf{Set}^{U^{l+m}}$, and the type $Q[\overline{\alpha},\overline{\varepsilon}]$ is modelled as $\widetilde{Q} = \widetilde{Q}_1 \times \cdots \times \widetilde{Q}_p \in \mathbf{Set}^{U^{l+m}}$. The expressions c are modelled as functions $c : U^l \to U^n$. The polynomial P_D for a Simple GADT with constants is

$$\boxed{\begin{tikzcd} U^n & k\{\widetilde{Q}\} \arrow[l, "{[d_1,\ldots,d_k]\circ k\pi}"'] \arrow[r, "\nabla_k"] & \{\widetilde{Q}\} \arrow[r, "c\pi_l\pi"] & U^n \end{tikzcd}}$$

This gives the polynomial functor $F_D : \mathbf{Set}^{U^n} \to \mathbf{Set}^{U^n}$

$$F_D X(\overline{\tau}) = \sum_{\substack{\overline{\alpha}\in U^l,\overline{\varepsilon}\in U^m \\ \overline{\tau}\equiv c(\overline{\alpha})}} \widetilde{Q}(\overline{\alpha},\overline{\varepsilon}) \times X(d_1(\overline{\alpha},\overline{\varepsilon})) \times \cdots \times X(d_k(\overline{\alpha},\overline{\varepsilon}))$$

Theorem 3.5 *For every Simple GADT D with constant types, the polynomial functor F_D has an initial algebra* D. *Moreover,* D *can be explicitly constructed as*

$$\mathsf{D}(u) = \bigcup_{i\in Nat} F_D^i(\mathbf{0})(u)$$

for every $u \in U^n$.

Proof It is well-known that there are adjoints: $\Sigma_h \dashv h^* \dashv \Pi_h$ (see §6.2). Thus, being left adjoints, $([d_1,\ldots,d_k]\pi)^*$ and $\Sigma_{c\pi_l\pi}$ preserve colimits (see *e.g.* (Mac Lane 1971)). Moreover, since Π_{∇_k} is finitary (see *e.g.* (Gambino and Kock 2010, Sec. 1.19)), the composite F_D preserves filtered colimits. Hence one can apply the Basic Lemma of (Smyth and Plotkin 1982) to construct the initial algebra. This is as stated because colimits in categories of indexed sets are given pointwise. □

Example 3.6 (**Well-scoped λ-terms**) Consider the GADT of well-scoped untyped λ-terms.

```
data Lam :: * -> * where
  Var :: Fin n -> Lam n
  App :: Lam n -> Lam n -> Lam n
  Abs :: Lam (S n) -> Lam n
```

This is a Simple GADT with constant types because it contains `Fin n`. Using $\mathsf{Fin} \in \mathbf{Set}^U$ as constructed in §3.1, we define the polynomials

$$\begin{tikzcd} Var = & U & \varnothing \arrow[l, "!"'] \arrow[r, "!"] & \{\mathsf{Fin}\} \arrow[r, "\pi"] & U \\ App = & U & 2U \arrow[l, "\nabla"'] \arrow[r, "\nabla"] & U \arrow[r, "\mathrm{id}"] & U \\ Abs = & U & U \arrow[l, "\mathtt{S}"'] \arrow[r, "\mathrm{id}"] & U \arrow[r, "\mathrm{id}"] & U \end{tikzcd}$$

Applying Def. 2.2 yields the polynomial functor $F : \mathbf{Set}^U \to \mathbf{Set}^U$

$$FX(n) = \mathsf{Fin}(n) + X(n)\times X(n) + X(\mathtt{S}\ \ n)$$

which is the same as the well-known one from (Fiore et al. 1999). The initial algebra Lam can be explicitly calculated by repeatedly applying F to $\mathbf{0} \in \mathbf{Set}^U$ as in the case of Fin.

Example 3.7 (**Nested datatype style λ-terms**) Bird and Paterson gave a slightly different representation of well-scoped λ-terms using a so-called nested datatype (Bird and Paterson 1999b). It can be presented as a GADT as follows.

```
data Incr :: * -> * where
  Zero :: Incr a
  Succ :: Incr a -> Incr a

data Lam :: * -> * where
  Var :: a -> Lam a
  App :: Lam a -> Lam a -> Lam a
  Abs :: Lam (Incr a) -> Lam a
```

An important difference with the previous representations is that the domain of `Var` is a type variable `a`. The use of this type variable is essential to get an element (*i.e.* de Bruijn variable) from an instantiated type. The interpretation of the type variable in `Var` is $[\![\alpha \vdash \alpha]\!] \in \mathbf{Set}^U$, hence

$$\begin{tikzcd} Var = & U & \varnothing \arrow[l, "!"'] \arrow[r, "!"] & \{[\![\alpha\vdash\alpha]\!]\} \arrow[r, "\pi"] & U \\ App = & U & 2U \arrow[l, "\nabla"'] \arrow[r, "\nabla"] & U \arrow[r, "\mathrm{id}"] & U \\ Abs = & U & U \arrow[l, "\mathtt{Incr}"'] \arrow[r, "\mathrm{id}"] & U \arrow[r, "\mathrm{id}"] & U \end{tikzcd}$$

These yield the polynomial functor $F_{\mathtt{Lam}} : \mathbf{Set}^U \to \mathbf{Set}^U$

$$F_{\mathtt{Lam}}X(\tau) = [\![\vdash \tau]\!] + X(\tau)\times X(\tau) + X(\mathtt{Incr}\ \ \tau)$$

Here we assume that U does not contain types of the form $\mathtt{Lam}\ \sigma$. This is practically unproblematic because the type variable `a` is usually expected to be instantiated with type level natural numbers. The reason for this exclusion is that $[\![\vdash\tau]\!]$ uses $\mu F_{\mathtt{Lam}}$ when $\tau = \mathtt{Lam}\ \sigma$. The complete solution to cope with this mutual reference may be formulated in the framework of indexed induction-recursion (Dybjer and Setzer 2006) or using the technique of nested inductive types (Abbott et al. 2004).

Example 3.8 (**Typed expressions**) Consider the GADT of typed expressions.

```
data Expr :: * -> * where
 Const  :: Int -> Expr Int
 IsZero :: Expr Int -> Expr Bool
 If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

We set $K_{\mathbb{Z}} = \{\mathbb{Z} \mid u \in U\} \in \mathbf{Set}^U$, and define $K_{Int}, K_{Bool} : U \to U$ by $K_{Int}(\tau) = \mathtt{Int}$, $K_{Bool}(\tau) = \mathtt{Bool}$. The polynomials are

$$\begin{tikzcd} Const = & U & \varnothing \arrow[l, "!"'] \arrow[r, "!"] & \{K_{\mathbb{Z}}\} \arrow[r, "K_{Int}\pi"] & U \\ IsZero = & U & U \arrow[l, "K_{Int}"'] \arrow[r, "\mathrm{id}"] & U \arrow[r, "K_{Bool}"] & U \\ If = & U & 3U \arrow[l, "{[K_{Bool},\mathrm{id},\mathrm{id}]}"'] \arrow[r, "\nabla_3"] & U \arrow[r, "\mathrm{id}"] & U \end{tikzcd}$$

These yield the polynomial functor $F : \mathbf{Set}^U \to \mathbf{Set}^U$

$$\begin{aligned} FX(\tau) = {} & (\tau \equiv \mathtt{Int}) \times \mathbb{Z} \\ & + (\tau \equiv \mathtt{Bool}) \times X(\mathtt{Int}) \\ & + X(\mathtt{Bool}) \times X^2(\tau). \end{aligned}$$

where $(\tau \equiv \sigma) \overset{def}{=} \{\star \mid \tau \equiv \sigma\}$; *i.e.* $(\tau \equiv \sigma)$ is the singleton set $\{\star\}$ if $\tau = \sigma$, and the empty set if $\tau \neq \sigma$. This is equivalent to the definition by pattern-matching

$$\begin{aligned} FX(\mathtt{Int}) &= \mathbb{Z} + X(\mathtt{Bool}) \times X^2(\mathtt{Int}), \\ FX(\mathtt{Bool}) &= X(\mathtt{Int}) + X(\mathtt{Bool}) \times X^2(\mathtt{Bool}), \\ FX(\tau) &= X(\mathtt{Bool}) \times X^2(\tau) \qquad \text{otherwise.} \end{aligned}$$

3.3 Positive GADTs

We finally consider the most general case, allowing the constructors to be parameterised by functions.

Positive GADT

$\texttt{data}\ D : *^n \to *\ \texttt{where}$

$$K : \forall \overline{\alpha}^l, \overline{\varepsilon}^m . Q[\overline{\alpha},\overline{\varepsilon}] \to (S_1[\overline{\alpha},\overline{\varepsilon}] \to D(d_1[\overline{\alpha},\overline{\varepsilon}])) \to \cdots \to (S_k[\overline{\alpha},\overline{\varepsilon}] \to D(d_k[\overline{\alpha},\overline{\varepsilon}])) \to D(c[\overline{\alpha}])$$

Assumption: D does not appear in Q and every S_i.

Each S_i is modelled as $\widetilde{S}_i \in \mathbf{Set}^{U^{l+m}}$. We define the function $d : \{\!\!\{\widetilde{Q} \times (\widetilde{S}_1 + \cdots + \widetilde{S}_k)\}\!\!\} \to U^n$ by $d((\overline{\alpha},\overline{\varepsilon}),t,(i,s)) = d_i(\overline{\alpha},\overline{\varepsilon})$, where each $d_1,\ldots,d_k$ is modelled as $d_i : U^{l+m} \to U^n$. We also define $p_1((\overline{\alpha},\overline{\varepsilon}),t,(i,s)) = ((\overline{\alpha},\overline{\varepsilon}),t)$ for $t \in \widetilde{Q}(\overline{\alpha},\overline{\varepsilon})$, $s \in \widetilde{S}_i(\overline{\alpha},\overline{\varepsilon})$, $1 \le i \le k$.

Considering the previous accounts, we let the polynomial for a Positive GADT D be given by

$$\begin{tikzcd} U^n & \{\!\!\{\widetilde{Q} \times (\widetilde{S}_1 + \cdots + \widetilde{S}_k)\}\!\!\} \arrow[l, "d"'] \arrow[r, "p_1"] & \{\!\!\{\widetilde{Q}\}\!\!\} \arrow[r, "c\pi_l\pi"] & U^n \end{tikzcd}$$

This gives the polynomial functor

$$\begin{aligned} F_D X(\overline{\tau}) &= \sum_{\substack{((\overline{\alpha},\overline{\varepsilon}),t) \in \{\!\!\{\widetilde{Q}\}\!\!\} \\ \overline{\tau} \equiv c(\overline{\alpha})}} \prod_{((\overline{\alpha},\overline{\varepsilon}),t,(i,s)) \in \{\!\!\{\widetilde{Q}\}\!\!\} \times (\widetilde{S}_1 + \cdots + \widetilde{S}_k)} X(d_i((\overline{\alpha},\overline{\varepsilon}),s)) \\ &= \sum_{\substack{\overline{\alpha} \in U^l, \overline{\varepsilon} \in U^m \\ \overline{\tau} \equiv c(\overline{\alpha})}} \widetilde{Q}(\overline{\alpha},\overline{\varepsilon}) \times (\widetilde{S}_1(\overline{\alpha},\overline{\varepsilon}) \to X(d_1(\overline{\alpha},\overline{\varepsilon}))) \times \cdots \\ &\qquad \times (\widetilde{S}_k(\overline{\alpha},\overline{\varepsilon}) \to X(d_k(\overline{\alpha},\overline{\varepsilon}))) \end{aligned}$$

These polynomial functors have weaker preservation properties than those of the previous cases: they preserve κ-colimits for κ a limit ordinal greater than the cardinality of every $\widetilde{S}_i(\overline{\alpha},\overline{\varepsilon})$ (so that, for instance, they preserve ω-colimits when every $\widetilde{S}_i(\overline{\alpha},\overline{\varepsilon})$ is a finite set). In this case, the existence and explicit construction of initial algebras is guaranteed by a transfinite version of the Basic Lemma (Smyth and Plotkin 1982) previously developed by (Adamek 1974). Hence we also have the following.

Proposition 3.9 *The polynomial functor of a Positive GADT has an initial algebra.*

4. Inductive Families as Polynomials

Dybjer introduced the notion of Inductive Family (IF) as a powerful schema for inductively defined dependent types (Dybjer 1994). For brevity, we treat here a slightly cut down version of this notion. As before, we do this in a stepwise fashion, considering first *Simple IFs* and then *Basic IFs*.

In this section, we use Agda notation (Norell 2007). Furthermore, we assume that there are no mutual definitions between IFs in a program. That is, any type appearing in the declaration of an IF does not refer to it.

4.1 Simple Inductive Families

We consider first a simple, yet practical, class of IFs close to Simple GADTs with constant types.

Simple Inductive Family

$\texttt{data}\ D : I \to \texttt{Set where}$

$$K :\ (j : J) \to (e : E) \to Q[j,e] \to D(d_1[j,e]) \to \cdots \to D(d_k[j,e]) \to D(c[j])$$

Here $I = I_1 \times \cdots \times I_n$, $J = J_1 \times \cdots \times J_l$, and $E = E_1 \times \cdots \times E_m$, $Q = Q_1 \times \cdots \times Q_q$.

Assumption: D does not appear in I, J, E, Q

We assume I, J, E to be product types to avoid clutter. Of course, our modelling works also for the usual presentation in curried form.

In a Simple IF declaration, E is a type other than D. Each $d_1,\ldots,d_k,c$ is a sequence of expressions of length n. Also $j : J$ denotes a sequence $(j_1,\ldots,j_l) : J_1 \times \cdots \times J_l$ (though we avoid the tedious notation $\overline{j}$ for it), as so does $e : E$.

To compare the scheme for Simple IFs to the one for Simple GADTs with constants, note that the variable j and the type E in a Simple IF respectively correspond to the type variables $\overline{\alpha}$ and the constant part Q in a Simple GADT.

The essential difference is that in a Simple IF the parameter j does not stand for a tuple of types but for a tuple of elements of types $J_1,\ldots,J_l$. This scheme is more natural than GADTs in many practical examples because one does not need to use a type class technique to express constraints.

The scheme of Simple IFs practically covers almost all everyday uses of IFs. Indeed, except for the use of large type parameters such as $(A : \texttt{Set})$, all examples in (Oury and Swierstra 2008) are Simple IFs.

Polynomial for a Simple IF. Our polynomial modelling follows.

- The types $J_1,\ldots,J_l$ are modelled as sets by means of the initial-algebra semantics, and $J \overset{def}{=} J_1 \times \cdots \times J_l \in \mathbf{Set}$. Likewise, $E \overset{def}{=} E_1 \times \cdots \times E_m \in \mathbf{Set}$.
- The type Q is modelled as an indexed set $Q \in \mathbf{Set}^{J \times E}$.
- Each expression $d_i[j,e]$ is modelled as a function $d_i : J \times E \to I$.
- The expression c is modelled as a function $c : J \to I$.

The polynomial P_D for this scheme is the same as the polynomial for Simple GADTs with constant types.

$$\begin{tikzcd} I & k\{\!\!\{Q\}\!\!\} \arrow[l, "{[d_1,\ldots,d_k] \circ k\pi}"'] \arrow[r, "\nabla_k"] & \{\!\!\{Q\}\!\!\} \arrow[r, "c\pi_l\pi"] & I \end{tikzcd}$$

It gives the polynomial functor $F_D : \mathbf{Set}^I \to \mathbf{Set}^I$

$$F_D X(m) = \sum_{\substack{j \in J, e \in E \\ m \equiv c(j)}} Q(j,e) \times X(d_1(j,e)) \times \cdots \times X(d_k(j,e))$$

Theorem 4.1 . *For every Simple IF D, the polynomial functor F_D has an initial algebra* D. *Moreover,* D *can be explicitly constructed as*

$$\mathsf{D}(u) = \bigcup_{i \in Nat} F_P^i(\mathbf{0})(u)$$

for every $u \in I$.

Proof Analogous to Thm. 3.5. □

4.2 Basic Inductive Families

We introduce an extension to the scheme of the previous section.

Basic Inductive Family

$$\begin{array}{l}\texttt{data}\ D : I \to \texttt{Set where}\\ \quad K : (j : J) \to\\ \qquad (e : E[j]) \to ((s : S_1[j,e]) \to D(d_1[j,e,s])) \to \cdots\\ \qquad\qquad \to ((s : S_k[j,e]) \to D(d_k[j,e,s])) \to D(c[j])\end{array}$$

where $I = I_1 \times \cdots \times I_n$, $J = J_1 \times \cdots \times J_l$, $E = E_1 \times \cdots \times E_m$.

Assumption: D does not appear in I, J, E and every S_i.

This scheme is similar to that of Positive GADTs (with the variable e corresponding to the type variables $\overline{\varepsilon}$) but generalised to constructors parameterised by dependent functions.

Polynomial for a Basic IF. Modelling Basic IFs is a further elaboration on the previous developments.

- J, c are modelled as before.
- E is modelled as an indexed set $E \in \mathbf{Set}^J$.
- Each type S_i is modelled as an indexed set $S_i \in \mathbf{Set}^{\{E\}}$; so that $S_1 + \cdots + S_k \in \mathbf{Set}^{\{E\}}$, and hence $\{S_1 + \cdots + S_k\} \in \mathbf{Set}$.
- Each expression $d_i[j,e,s]$ is modelled as a function $d_i : \{S_i\} \to I$.
- The function $d : \{S_1 + \cdots + S_k\} \to I$ is defined by
$$d((j,t),(i,s)) = d_i(j,t,s)$$

The polynomial P_D for a Basic IF D is

$$\boxed{I \xleftarrow{d} \{S_1 + \cdots + S_k\} \xrightarrow{\pi} \{E\} \xrightarrow{c\pi} I}$$

This gives the polynomial functor $F_D : \mathbf{Set}^I \to \mathbf{Set}^I$

$$\begin{aligned} F_P X(u) &= \sum_{\substack{j \in J\\ u \equiv c(j)}} \prod_{(i,s) \in (S_1 + \cdots + S_k)(j,e)} X(d_i(j,e,s))\\ &\cong \sum_{\substack{j \in J, e \in E(j)\\ u \equiv c(j)}} \prod_{s \in S_1(j,e)} X(d_1(j,e,s)) \times \cdots \times \prod_{s \in S_k(j,e)} X(d_k(j,e,s)) \end{aligned}$$

As in Prop. 3.9 we have the following.

Proposition 4.2 *The polynomial functor for a Basic IF has an initial algebra.*

Basic IFs are very similar to the Inductive Families of (Dybjer 1994). The differences are that in the latter

(i) every sequence of types is a so-called telescope (namely, later types may depend on earlier types in the sequence), and

(ii) the large type `Set` is allowed as a type of a parameter.

The model above can be extended to cope with these: (i) is easily modelled by using the dependent sum of sets corresponding to the types; as for (ii), modelling large types can be treated in the framework of induction-recursion (Dybjer and Setzer 2003, 2006).

5. Application: Dependent Zippers

As an application of the use of polynomial functors, we consider zippers for IFs.

A zipper is a datatype for navigating a tree-like structure (Huet 1997). It is given by a pair of the current focus (called the *cursor*) in a tree, and a context consisting of the list of depth-one linear contexts obtained after navigation. Zippers have only been considered on ordinary ADTs (Huet 1997; McBride 2001; Adams 2010; Morihata et al. 2009), such as lists and trees.

McBride found the interesting phenomenon that the derivative of the polynomial functor associated with an ADT provides the type of depth-one linear contexts for the zipper on the datatype (McBride 2001). For instance, for the ADT of binary trees with associated polynomial functor $FX = 1 + X \times X$, the formal derivative $F'X = X + X$ gives the type of depth-one linear contexts because: when the cursor goes to the left (resp. right) child of a tree, one needs to push the right (resp. left) subtree to a stack for later navigation; so that the left (resp. right) summand in $F'X$ is for the right (left) depth-one linear context.

In this section, we consider *partial derivatives* for dependent polynomial functors, observe that the notion is again well-suited for the representation of depth-one linear contexts, and thereby develop *zippers on IFs and GADTs*.

5.1 Partial derivatives of the polynomial functor for λ-terms

We saw in Example 3.6 that the polynomial functor for well-scoped λ-terms is $F : \mathbf{Set}^{Nat} \to \mathbf{Set}^{Nat}$ defined by

$$FX(n) = \mathsf{Fin}(n) + X(n) \times X(n) + X(\mathtt{S}\,n).$$

where *Nat* is the ADT of natural numbers.

We now want to built the zipper for λ-terms. This is of practical importance; for instance, to develop programs for navigating and modifying λ-terms (in a structured editor, in an interpreter, *etc.*) represented by the GADT or IF of this polynomial functor. To define the zipper, we need the type of depth-one linear contexts. This we will build from the partial derivatives of the polynomial functor.

For the case of well-scoped λ-terms, the partial derivative of F with respect to $m \in Nat$ is $\partial_m F : \mathbf{Set}^{Nat} \to \mathbf{Set}^{Nat}$ given by

$$\partial_m F(X)(n) = (m \equiv n) \times (X(n) + X(n)) + (m \equiv \mathtt{S}\,n) \times 1. \quad (2)$$

In general, the index m of the partial derivative $\partial_m F$ designates the index of the cursor of the zipper. In the case of the GADT or IF of λ-terms, an index of the datatype is an environment (*i.e.* a set of all possible free variables). The summands of (2) are the type of depth-one linear contexts in the environment n. At the technical level one can see this by observing that there is a canonical function

$$\mathsf{plug} : \mu F(m) \times \partial_m F(\mu F)(n) \to \mu F(n)$$

that plugs a λ-term in environment m into a depth-one linear context for λ-terms in environment n to construct a λ-term in environment n. Indeed, according to the three summands of (2), the function plug is built from the following three functions

$$\begin{array}{ll} \mathsf{plugL} : \mu F(m) \times \mu F(m) \to \mu F(m) & ,\ \mathsf{plugL}(t,c) = \mathtt{App}\ t\ c\\ \mathsf{plugR} : \mu F(m) \times \mu F(m) \to \mu F(m) & ,\ \mathsf{plugR}(t,c) = \mathtt{App}\ c\ t\\ \mathsf{plugI} : \mu F(\mathtt{S}\,m) \to \mu F(m) & ,\ \mathsf{plugI}(t) = \mathtt{Abs}\ t \end{array}$$

that respectively plug a term in the left-hand-side of an application, in the right-hand-side of an application, and inside an abstraction.

Formula (2) is obtained as an instance of a general formula for partial derivation that we introduce next.

5.2 Differentiation

The differentiation operator ∂ maps a polynomial functor $F : \mathbf{Set}^I \to \mathbf{Set}^J$ to a polynomial functor $\partial F : \mathbf{Set}^I \to \mathbf{Set}^{I \times J}$. It is assembled from the partial derivatives of F with respect to $i \in I$, for which we write $\partial_i F : \mathbf{Set}^I \to \mathbf{Set}^J$, in a pointwise fashion as follows

$$\partial F(X)(i,j) = \partial_i F(X)(j).$$

$$\begin{array}{l}
\texttt{data Ctx} : I \to \texttt{Set where} \\
\quad \texttt{[]} : \{m : I\} \to \texttt{Ctx}(m) \\
\left\{ \begin{array}{l} K_i : \{j : J\} \to \{e : E\} \to Q[j,e] \to \\ \qquad \underbrace{D(d_1[j,e]) \to \cdots \to D(d_k[j,e])}_{\text{with only } D(d_i[j,e]) \text{ missing}} \to \texttt{Ctx}(c[j]) \to \texttt{Ctx}(d_i[j,e]) \end{array} \right\}_{i \in [k]}
\end{array}$$

$$\begin{array}{l}
\texttt{data Zipper} : I \to \texttt{Set where} \\
\quad \_ \triangleright \_ : \{m : I\} \to D(m) \to \texttt{Ctx}(m) \to \texttt{Zipper}(m) \\
\quad \texttt{failure} : \{m : I\} \to \texttt{Zipper}(m)
\end{array}$$

$$\begin{array}{l}
\texttt{down}_i : \{j : J\} \to \{e : E\} \to \texttt{Zipper}(c[j]) \to \texttt{Zipper}(d_i[j,e]) \\
\texttt{down}_i(K\ q\ a_1 \ldots a_{i-1}\ a_i\ a_{i+1} \ldots a_k\ \triangleright\ C) = (a_i\ \triangleright\ K_i\ q\ a_1 \ldots a_{i-1}\ a_{i+1} \ldots a_k\ C) \\
\texttt{upfrom}_i : \{j : J\} \to \{e : E\} \to \texttt{Zipper}(d_i[j,e]) \to \texttt{Zipper}(c[j]) \\
\texttt{upfrom}_i(a_i\ \triangleright\ K_i\ q\ a_1 \ldots a_{i-1}\ a_{i+1} \ldots a_k\ C) = (K\ q\ a_1 \ldots a_{i-1}\ a_i\ a_{i+1}\ a_k\ \triangleright C)
\end{array}$$

Figure 1. Zipper for a Simple IF D

The partial derivative operators are to satisfy the following basic laws.

- For every monomial $M = (I \xleftarrow{d} E \xrightarrow{!} 1 \xrightarrow{\mathrm{id}} 1)$ with induced monomial functor $F_M(X) = \prod_{e \in E} X(d(e))$,
$$\partial_i F_M(X) \;\cong\; \sum_{\substack{\ell \in E \\ i \equiv d(\ell)}} \prod_{e \in E \setminus \{\ell\}} X(d(e)).$$
- For a family of polynomial functors $\{F_k : \mathbf{Set}^I \to \mathbf{Set}^J \mid k \in K\}$,
$$\partial_i \left(\textstyle\sum_{k \in K} F_k\right)(X) \cong \textstyle\sum_{k \in K} \partial_i F_k(X).$$

It follows that for a polynomial $P = (I \xleftarrow{d} E \xrightarrow{p} B \xrightarrow{c} J)$ with induced polynomial functor

$$F_P(X)(j) = \sum_{\substack{b \in B \\ j \equiv c(b)}} \prod_{\substack{e \in E \\ b \equiv p(e)}} X(d(e))$$

its partial derivative should be given by

Partial derivatives of polynomial functors

$$\partial_i F_P(X)(j) = \sum_{\substack{b \in B \\ j \equiv c(b)}} \sum_{\substack{\ell \in E_b \\ i \equiv d(\ell)}} \prod_{e \in E_b \setminus \{\ell\}} X(d(e)) \qquad (3)$$

where

$$E_b \overset{def}{=} \{e \in E \mid b \equiv p(e)\}$$

Intuitively, this definition of partial derivative can be understood as follows. The set E_b consists of the available indices of the product part of the polynomial functor F_P. If the index of the cursor of a zipper uses $\ell \in E_b$, then the indices of the rest depth-one linear context are the elements in E_b except for ℓ (as this index is being used as that for the cursor).

We instantiate the general formula for partial derivatives in two concrete cases.

Proposition 5.1 *The partial derivatives of the polynomial functor for a Simple IF D are*

$$\partial_m F_D(X)(n) \;\cong\; \sum_{\substack{j \in J, e \in E \\ n \equiv c(j)}} \sum_{\substack{i \in [k] \\ m \equiv d_i(j,e)}} Q(j,e) \times \prod_{\ell \in [k] \setminus \{i\}} X(d_\ell(j,e))$$

where $[k] \overset{def}{=} \{\ell \in \mathbb{N} \mid 1 \le \ell \le k\}$.

The partial derivatives of the polynomial functor for λ-terms of (2) are an instance of the above proposition.

Proposition 5.2 *The partial derivatives of the polynomial functor for a Basic IF D are*

$$\partial_m F_D(X)(n) = \sum_{\substack{j \in J, e \in E(j) \\ n \equiv c(j)}} \sum_{\substack{(i,s) \in S(j,e) \\ m \equiv d_i(j,e,s)}} \prod_{(i',s') \in S(j,e) \setminus \{(i,s)\}} X(d_{i'}(j,e,s'))$$

where $S \overset{def}{=} S_1 + \cdots + S_k$.

Polynomials are indeed closed under differentiation; as one has that

$$\partial F_P(X) \cong F_{\partial P}(X)$$

for

$$\partial(I \xleftarrow{d} E \xrightarrow{p} B \xrightarrow{c} J) \overset{def}{=} (I \xleftarrow{d'} E' \xrightarrow{p'} B' \xrightarrow{c'} I \times J)$$

where

$$\begin{aligned}
E_{i,b} &= \{\ell \in E \mid i \equiv d(\ell), b \equiv p(\ell)\}\ , \\
E' &= \sum_{(i,j) \in I \times J} \sum_{\substack{b \in B \\ j \equiv c(b)}} \sum_{\ell \in E_{i,b}} E_b \setminus \{\ell\}\ , \\
B' &= \sum_{(i,j) \in I \times J} \sum_{\substack{b \in B \\ j \equiv c(b)}} E_{i,b}\ , \\
&d'((i,j),b,\ell,e) = d(e)\ , \\
&p'((i,j),b,\ell,e) = ((i,j),b,\ell)\ , \\
&c'((i,j),b,e) = (i,j)\ .
\end{aligned}$$

Finally, and importantly for what follows, we note that for a polynomial functor $F : \mathbf{Set}^I \to \mathbf{Set}^I$, its partial derivatives come equipped with canonical functions

(i) $\mu F(m) \times \partial_m F(\mu F)(n) \to \mu F(n)$, and

(ii) $\mu F(n) \times \sum_{b \in B} E_{m,b} \to 1 + \big(\mu F(m) \times \partial_m F(\mu F)(n)\big)$

for (i) plugging-in data in a depth-one linear context, and (ii) for, whenever possible, disassembling data at a given index, into a datum and a depth-one linear context.

5.3 Dependent zippers

The zipper for a polynomial functor $F : \mathbf{Set}^I \to \mathbf{Set}^I$ is the indexed set $Zipper \in \mathbf{Set}^I$ defined as

$$Zipper(m) \overset{def}{=} \mu F(m) \times Ctx(m)$$

where

$$Ctx(m) \cong 1 + \sum_{n \in I} \partial_m F(\mu F)(n) \times Ctx(n). \qquad (4)$$

As such, *Zipper* consists of pairs of the current focus and a context. A context in *Ctx* is a list of depth-one linear contexts for the focus.

Simple IFs. Figure 1 gives an implementation of the zipper datatype for a Simple IF D in Agda. It is based on the partial derivatives of the associated polynomial functor F_D as explained above, and provides the navigation operations `down` and `upfrom`.

The implementation of *Ctx* by means of the datatype `Ctx` is derived as follows. By (4), the type of a constructor of `Ctx` should be

$$\sum_{n \in I} \partial_m F_D(D)(n) \times Ctx(n) \to Ctx(m).$$

By expanding $\partial_m F_D(D)(n)$ for the case of the Simple IF D (see Prop. 5.1), this type is isomorphic to

$$\sum_{n \in I} \sum_{\substack{j \in J, e \in E \\ n \equiv c(j)}} \sum_{\substack{i \in [k] \\ m \equiv d_i(j,e)}} Q(j,e) \times \prod_{\ell \in [k] \setminus \{i\}} D(d_\ell(j,e)) \times Ctx(n) \to Ctx(m).$$

By instantiating the equality constraints for n and m, and swapping the second and the third sum, this is in turn isomorphic to

$$\sum_{i \in [k]} \sum_{\substack{j \in J \\ e \in E}} Q(j,e) \times \prod_{\ell \in [k] \setminus \{i\}} D(d_\ell(j,e)) \times Ctx(c(j)) \to Ctx(d_i(j,e))$$

which is the type used for the constructor K_i in the declaration of `Ctx` in Figure 1. Thus, we implement *Ctx* as the datatype `Ctx` with depth-one linear contexts given by constructors K_i, where i indicates the missing argument of the original constructor K.

The navigation operations are defined schematically for every K and K_i. Failure cases are omitted for simplicity. The function ($\texttt{down}_i\ z$) expresses "going down to the i-th child in the D-part in the zipper z". Similarly, ($\texttt{upfrom}_i\ z$) expresses "going up from the i-th child". Going down and up to a child in the Q-part in a zipper are defined similarly; as so are change, insertion, and deletion operations on zippers.

Instantiating this general zipper for the datatype of λ-terms, we obtain the zipper on well-scoped λ-terms. The zipper at the root of a λ-term t is ($t \triangleright$ `[]`), expressing that the cursor is focused on t and that the current context is empty. Starting from this, the navigation operations enable any traversal of the structure of the λ-term. In this process, a context increases or decreases as needed. Note that the crucial operations of stripping and adding binders on λ-terms by means of `down` and `upfrom` are defined using suitable indexes on the type `Zipper`. Hence the operations maintain the invariant of terms being well-scoped. This is one of the principles of dependently-typed programming: program correctness is ensured by typing.

Note also that our schema is *generic* in the sense that its instantiation to any Simple IF yields a zipper. We say that a datatype is *zipperable* whenever there is a zipper datatype for it that supports navigation operations. As we have seen, then, every Simple IF is zipperable.

A Simple IF can be simulated by a Simple GADT with constant types because the polynomials obtained for these are the same. This is an example of what we have referred to in the Introduction as "enabling the transfer of technologies (the zipper) developed in one world (IFs) to another one (GADTs)". It follows that every Simple GADT is zipperable.

Basic IFs. Is a Basic IF zipperable? In theory, yes; though in practice this is unclear. This is because even though we have obtained an explicit formula for the partial derivatives of arbitrary Basic IFs from which one can in principle implement the type `Zipper` together with the navigation operations, the kind of dependently-typed programming required may be too involved. The issue here is the implementation of the "set difference" $S(j,e) \setminus \{(i,s)\}$ used in the definition of $\partial_m F(X)(n)$ (see Prop. 5.2). This operation invokes computations and equality in the definition of the datatype `Ctx` with the effect that the pattern matching in the `down` and `upfrom` operations requires explicit proofs to be passed to the type checker. (This complication does not appear in the case of Simple IFs because the required "set difference" was implemented by explicitly omitting an argument from the type of a context K_i in the definition of `Ctx`.)

Fortunately, almost all dependently-typed data structures found in the literature (Oury and Swierstra 2008; Norell 2008; Bove and Dybjer 2008) are Simple IFs. Hence there seems to be no current need for zippers on non-simple inductive families. We hope that the difficulty discussed above will be resolved when dependently-typed programming is mature.

In the case of Positive GADTs the situation is worse, as Haskell seems not to be powerful enough to manipulate the needed complex types for indexing.

6. Discussion and Related Work

6.1 Henry Ford encoding and GADTs

The departure of GADTs from ordinary ADTs resides in that the codomain of each constructor may have various instances, *e.g.*

```
data Expr :: * -> * where
 Const  :: Int -> Expr Int
 IsZero :: Expr Int -> Expr Bool
 If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Using equality constraints, however, this can be rewritten as

```
data Expr :: * -> * where
 Const  :: (a ~ Int)  => Int -> Expr a
 IsZero :: (a ~ Bool) => Expr Int -> Expr a
 If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

where (`a ~` τ) is the equality constraint $\mathtt{a} \equiv \tau$ on types. In this manner, GADTs can be regarded as parameterised ordinary ADTs, with all constructors having the same codomain type, but under equality constraints. This technique is known from the early days of GADTs (Hinze 2003) and has been used in the formulation of type inference for GADTs (Schrijvers et al. 2009).

In fact, the idea was already known for IFs before GADTs were proposed by the Haskell community. McBride explained the general technique using Henry Ford's phrase: "*You can have any color you like, as long as it's black.*" (McBride 1999, Sec. 3.5). In this light, the use of the equality type in a constructor declaration such as the one below

$$Const : \forall \alpha.(\alpha \equiv Int) \to Int \to Expr(\alpha)$$

expresses that "you can have any type α you like, as long as it's *Int*".

6.2 Left Kan extensions and GADTs

Another technique to make all codomains of a GADT declaration have the same type (by appropriately shifting the varying types to the domain side) involves the use of the category-theoretic concepts of Kan extension and adjunction[1]. These notions play a fundamental role in what regards to the three functors used to define

[1] In the context of functional programming, the concept of right Kan extension has been used to derive generalised fold for nested datatypes (Bird and Paterson 1999a; Abel et al. 2005). Various adjunctions, including those for Kan extensions, are applied to develop "adjoint fold" techniques in (Hinze 2010).

polynomial functors, and so we review that first. Subsequently, we consider encodings for GADTs by left Kan extensions.

Kan extensions. It is well-known in categorical logic and type theory that the dependent sum functor Σ_h, the reindexing functor h^*, and the dependent product functor Π_h form a string of adjunctions. Explicitly, for every h, we have the following two adjoint pairs of functors (Lawvere 1969)

$$\Sigma_h \dashv h^* \dashv \Pi_h .$$

This fact was used in arguing for the existence of initial algebras.

This adjointness is an instance of another well-known adjointness characterising *Kan extensions*:

$$\mathrm{Lan}_h \dashv (-\circ h) \dashv \mathrm{Ran}_h .$$

Here Lan_h, *left Kan extending* along h, is the left adjoint to the precomposition functor $(-\circ h) = h^*$. Dually, Ran_h, *right Kan extending* along h, is the right adjoint to h^*.

In the presence of enough categorical structure, left and right Kan extending can be respectively described by means of coends and ends (Mac Lane 1971, Chap. X.4), for which one typically uses the notation $\int$ due to Yoneda (Yoneda 1960). (A coend is a colimit, readily described as a coequalizer between coproducts; dually, an end is a limit, readily described as an equalizer between products.) In the context of indexed sets, the (co)end formulae for Kan extending along $h : I \to J$ are:

$$(\mathrm{Lan}_h X)(j) = \int^{i\in I} J(h(i),j)\times X(i) \;\cong\; \sum_{\substack{i\in I\\ j\equiv h(i)}} X(i) = \Sigma_h(X)(j)$$

$$(\mathrm{Ran}_h X)(j) = \int_{i\in I} [J(j,h(i)) \Rightarrow X(i)] \;\cong\; \prod_{\substack{i\in I\\ j\equiv h(i)}} X(i) = \Pi_h(X)(j)$$

Note that a set J is considered as a discrete category. Hence the homset $J(h(i), j)$ is either empty or consists only of the identity, which entails the equality constraining $j \equiv h(i)$.

Lan encoding for GADTs. Johann and Ghani have studied a technique to provide initial-algebra semantics for GADTs using left Kan extensions (Johann and Ghani 2008). Their approach is to reduce the problem to the case of ordinary initial-algebra semantics of ADTs. For example, in this theory, the GADT `Fin` is transformed to the following one

```
data Fin :: * -> * where
  NZero :: Lan S One j -> Fin j
  NSucc :: Lan S Fin j -> Fin j

data One j      = One
data Lan h _X j = forall i. Lan (Eql j (h i), _X i)
data Eql i j    where Refl :: Eql i i
```

This seems mysterious; though, as we proceed to explain, the transformed datatype is equivalent to the original one.

First, the type `Lan` is an implementation of the left Kan extension formula for $\mathrm{Lan}_h X(j)$ as $\sum_{i\in U}(j \equiv h(i)) \times X(i)$. The set-theoretic sum with equality is implemented by an existential type (NB: this `forall` means $\exists$) with the equality type `Eql` on Haskell types.

Second, since Lan_h is left adjoint to $-\circ h$, there is a one-to-one correspondence as follows

$$\frac{f : \mathsf{Fin} \to \mathsf{Fin}\circ \mathtt{S}}{\check{f} : \mathrm{Lan}_{\mathtt{S}}\, \mathsf{Fin} \to \mathsf{Fin}} \qquad (5)$$

That is, precomposition with `S` in the codomain can be transported to left Kan extension along `S` in the domain[2]. Hence, by adjointness, to give

$$\mathit{NSucc} : (\mathrm{Lan}_{\mathtt{S}}\,\mathsf{Fin})(j) \to \mathsf{Fin}(j)$$

is mathematically equivalent to give

$$\mathit{Succ} : \mathsf{Fin}(j) \to \mathsf{Fin}(\mathtt{S}(j)).$$

The Lan encoding is thus equivalent to the original definition.

6.3 Henry Ford and Lan encodings

We observe that the Henry Ford and Lan encodings are equivalent. Indeed, using that $(\mathrm{Lan}_{\mathtt{S}}\,\mathsf{Fin})(j) \cong \sum_{i\in U, j\equiv \mathtt{S}(i)} \mathsf{Fin}(i)$, one equivalently has that

$$\mathit{NSucc} : \sum_{\substack{i\in U\\ j\equiv \mathtt{S}(i)}} \mathsf{Fin}(i) \to \mathsf{Fin}(j).$$

This is a set-theoretical expression of the Henry Ford encoding

```
Succ :: (j ~ S i) => Fin i -> Fin j
```

stating "you can have any j you like, as long as it's $\mathtt{S}(i)$ (for some i)".

6.4 Polynomial functor approach

The polynomial functor approach structurally subsumes the Henry Ford and Lan approaches. We explain this in the light of the previous example.

Recall that the polynomial associated to the constructor

```
Succ :: Fin a -> Fin (S a)
```

is

$$\mathit{Succ} = \begin{tikzcd} U & U \arrow[l, "\mathrm{id}"'] \arrow[r, "\mathrm{id}"] & U \arrow[r, "\mathtt{S}"] & U \end{tikzcd} .$$

Its associated polynomial can be presented in the following three equivalent forms

$$\underbrace{\sum_{i\in U, j\equiv \mathtt{S}(i)} X(i)}_{=\,\Sigma_{\mathtt{S}} X(j)} \;\cong\; \underbrace{\sum_{i\in U} (j\equiv \mathtt{S}(i))\times X(i)}_{\cong\,\mathrm{Lan}_{\mathtt{S}} X(j)} \;\cong\; \underbrace{\Sigma_{\mathtt{S}}\Pi_{\mathtt{id}}\mathrm{id}^*(X)(j)}_{=\,F_{\mathit{Succ}} X(j)} .$$

It is from these that the three approaches to modelling the type of `Succ` arise; namely

(Henry Ford encoding)	$\sum_{i\in U, j\equiv \mathtt{S}(i)} \mathsf{Fin}(i) \to \mathsf{Fin}(j)$
(Lan Encoding)	$\mathrm{Lan}_{\mathtt{S}}\,\mathsf{Fin}(j) \to \mathsf{Fin}(j)$
(Polynomial Functor)	$F_{\mathit{Succ}}\mathsf{Fin}(j) \to \mathsf{Fin}(j)$

The Henry Ford and Lan encodings are just ways in which to transfer indexing structure from the codomain to the domain. As such, then, they correspond to the polynomial functor approach restricted to polynomials of the form

$$\begin{tikzcd} I & I \arrow[l, "\mathrm{id}"'] \arrow[r, "\mathrm{id}"] & I \arrow[r, "c"] & J \end{tikzcd} .$$

What is the advantage of the polynomial functor approach? There are at least three advantages.

[2] This is the fundamental principle underlying adjunctions (and Galois connections). Currying may be the most well-known adjunction for functional programmers:

$$\frac{f : A \to (B \Rightarrow C)}{\check{f} : A\times B \to C} \qquad (6)$$

Observe the formal similarity between (5) and (6).

[I] The first advantage resides in the general form

$$\begin{tikzcd} I & E \arrow[l, "d"'] \arrow[r, "p"] & B \arrow[r, "c"] & J \end{tikzcd}$$

taken by *polynomials*, that allows one to express varying multi-ary forms for the argument types of a constructor. This has been a main topic of the paper. For instance, we saw in §3.1 that Simple GADTs are modelled by a polynomial of the form

$$\begin{tikzcd} U^n & kU^{l+m} \arrow[l, "{[d_1,\ldots,d_k]}"'] \arrow[r, "\nabla_k"] & U^{l+m} \arrow[r, "c\pi_l"] & U^n \end{tikzcd} .$$

An algebra for the associated polynomial functor is an indexed set X together with an indexed function

$$\sum_{\substack{i\in U\\ j\equiv c(i)}} X(d_1(i)) \times \cdots \times X(d_k(i)) \to X(j)$$

that provides a k-ary constructor with varying indexing as specified by each of the d_i. As shown in §3.3 and §4.2, more involved forms for the domain (*e.g.* with dependent function spaces) can be also expressed.

[II] The second more important advantage is that every polynomial functor has an *initial algebra* constructed by (a possibly transfinite) iteration. This property is not enjoyed by the Henry Ford and Lan approaches. In them initial algebras are only guaranteed when the arguments to Σ or Lan are well-behaved. Hence the polynomial functor approach is generic, and more appropriate than the other two approaches for modelling GADTs and IFs; making clear the polynomial character of these datatypes.

[III] The third prospective advantage is the *generality* of the framework. The original notions of polynomial and polynomial functor (Gambino and Hyland 2003) are more general that considered here and were developed within an ambient lccc (locally cartesian closed category), *viz.* a category for which every slice category is cartesian closed. The slice categories of the ambient lccc are used for indexing, and polynomial functors are induced by means of adjoints $\Sigma_h \dashv h^* \dashv \Pi_h$ between slices available for every map h. The set-theoretic setting is the special case for the lccc **Set**.

While in this paper we have restricted attention to the set-theoretic setting, it is interesting to consider domain-theoretic settings (*i.e.* categories of CPOs) and more involved models of higher-order polymorphism that more tightly match with Haskell. In this respect, lcccs may not be general enough and the more general *closed comprehension categories* may be more appropriate (Jacobs 1999, Sec. 10.6). These cover the domain-theoretic case and model dependent sums, reindexing, dependent products, and comprehension. As such, thus, they support our polynomial modelling of GADTs and IFs. This direction will be pursued in future work.

6.5 On comprehension

The construction of the set $\{Q\}$ for an indexed set Q of §3.2 can be understood from various categorical viewpoints. While this fact has not been used explicitly in the paper, it might be useful for further generalisation and so we note it here. For presheaves, this construction was first used by Yoneda and is often called the category of elements. More generally, it is also seen as the Grothendieck construction of an indexed category with the projection π forming a fibration. The notation $\{-\}$ is taken from that for "comprehension" in a comprehension category (Jacobs 1999, Sec. 10.4.7).

6.6 Coinductive datatypes

We have chosen the set-theoretic setting in this paper to concretely show the correspondence between datatype declarations, polynomials, and polynomial functors with minimal categorical notions, and aiming at the application to generic programming. This setting does not only suffices to model total functional programming with inductively defined datatypes, but also with coinductively defined ones. Indeed, since d^* and Π_p preserve limits, and Σ_c preserves limits of ω^{op}-chains, it follows that every polynomial functors has a final coalgebra. These are non-well-founded data structures. Hence, the lazy datatypes of Haskell and the codata feature of Agda might be modelled in this setting. The details are left as future work.

6.7 Further related work

Gambino and Hyland introduced dependent polynomial functors between slices of an ambient lccc and show that they have initial algebras provided that the ambient category admits W-types (Gambino and Hyland 2003). In this paper, we reveal the polynomials for GADTs and IFs, and thereby provide a simple and direct initial-algebra semantics by means of dependent polynomial functors that generalise the polynomial functors for ADTs. This characterisation is well-suited for exchanging technologies between functional programming and type theory.

The mathematics of polynomial functors is investigated in (Kock 2009; Gambino and Kock 2010). This work might be useful for further applications to GADTs and IFs with our analysis.

Inductive Families were formulated by (Dybjer 1994). (Dybjer and Setzer 2003) gave an initial-algebra characterisation of inductive-recursive definitions, which covers all IFs. Since one needs to model an extra *inductive-recursive* feature, the form of functors for IFs is not that clear in that presentation.

Indexed containers (Altenkirch and Morris 2009) are a type-theoretic rendering of dependent polynomial functors. Morris and Altenkirch gave a type-theoretic characterisation showing that indexed containers can express IFs.

Differentiation has been considered for generalised species of structures (Fiore 2005) and for (non-indexed) containers (Abbott et al. 2005). In both cases, derivatives can be characterised as linear exponentials. This extends to dependent polynomial functors between categories of indexed sets.

There are a number of works on generic programming *within* dependent type systems based on modelling dependent types in a dependent type theory (Altenkirch and McBride 2002; Benke et al. 2003; Chapman et al. 2010). These models are very often founded on some mathematical semantics of dependent type theory, such as (indexed) induction-recursion (Dybjer and Setzer 2003, 2006) or containers (Abbott et al. 2005; Altenkirch and Morris 2009), implementing these semantics at the level of type theory. The polynomial functor approach will be also realised in this way. This will serve as a basis for transporting generic functional programming techniques to dependent type theory.

Acknowledgments

We thank Tarmo Uustalu and Varmo Vene for discussions related to a shapely typed zipper in Agda, and Makoto Takeyama for information about the Henry Ford encoding. Makoto Hamana is supported in part by the JSPS Grant-in-Aid for Scientific Research (22700004).

References

M. Abbott, T. Altenkirch, and N. Ghani. Representing nested inductive types using W-types. In *ICALP'04*, pages 59–71, 2004.

M. Abbott, T. Altenkirch, C. McBride, and N. Ghani. ∂ for data: Differentiating data structures. *Fundam. Inform.*, 65(1-2):1–28, 2005.

A. Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.*, 333(1-2):3–66, 2005.

J. Adamek. Free algebras and automata realizations in the language of categories. *Comment. Math. Univ. Carolina*, 15(589602), 1974.

M. D. Adams. Scrap your zippers: A generic zipper for heterogeneous types. In *WGP '10*, pages 13–24. ACM, 2010.

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Generic Programming*, pages 1–20, 2002.

T. Altenkirch and P. Morris. Indexed containers. In *LICS'09*, pages 277–285, 2009.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

R. Bird and R. Paterson. Generalised folds for nested datatypes. *Form. Asp. of Comput.*, 11(2):200–222, 1999a.

R. Bird and R. Paterson. De Bruijn notation as a nested datatype. *J. of Funct. Program.*, 9(1):77–91, 1999b.

A. Bove and P. Dybjer. Dependent types at work. In *LerNet ALFA Summer School*, pages 57–99, 2008.

J. Chapman, P. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *Proc. of ICFP'10*, pages 3–14, 2010.

N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Proc. of POPL'06*, pages 206–217, 2006.

P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1994.

P. Dybjer and A. Setzer. Indexed induction-recursion. *J. Log. Algebr. Program.*, 66(1):1–49, 2006.

P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Ann. Pure Appl. Logic*, 124(1-3):1–47, 2003.

M. Fiore. Mathematical models of computational and combinatorial structures. In *Proc. FOSSACS 2005*, LNCS 3441, pages 25–46, 2005.

M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of 14th Annual Symposium on Logic in Computer Science*, pages 193–202, 1999.

N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In *TYPES'03*, pages 210–225, 2003.

N. Gambino and J. Kock. Polynomial functors and polynomial monads. ArXiv:0906.4931, 2010.

N. Ghani, P. Johann, T. Uustalu, and V. Vene. Monadic augment and generalised short cut fusion. In *Proc. of ICFP'05*, pages 294–305, 2005.

T. Hagino. A typed lambda calculus with categorical type constructors. In Pitt et al., editor, *Category Theory in Computer Science'87*, LNCS 283, pages 140–157. Springer-Verlag, 1987.

R. Hinze. Fun with phantom types. In *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003.

R. Hinze. Adjoint folds and unfolds. In *Proc. of MPC'10*, pages 195–228, 2010.

R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In *Generic Programming*, LNCS 2793, pages 1–56, 2003.

Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proc. of ICFP'96*, pages 73–82, 1996.

G. Huet. Functional pearl: The zipper. *Journal of Functional Programming*, 6(5):549–554, 1997.

B. Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. North Holland, Elsevier, 1999.

P. Jansson and J. Jeuring. Polyp - a polytypic programming language. In *POPL*, pages 470–482, 1997.

P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *Proc. of POPL'08*, pages 297–308, 2008.

S. Katsumata and S. Nishimura. Algebraic fusion of functions with an accumulating parameter and its improvement. *J. Funct. Program.*, 18 (5-6):781–819, 2008.

J. Kock. Notes on polynomial functors. Manuscript, version 2009-08-05, 2009.

J. Launchbury and T. Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In *Proc. of FPCA'95*, pages 314–323, 1995.

F.W. Lawvere. Adjointness in foundations. *Dialectica*, pages 281–296, 1969.

S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer-Verlag, 1994.

C. McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, pages 130–170, 2004.

C. McBride. The derivative of a regular type is its type of one-hole contexts. Manuscript, 2001.

C. McBride. *Dependently Typed Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc of FPCA'91*, pages 124–144, 1991.

A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In *POPL'09*, pages 177–185, 2009.

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

U. Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266, 2008.

N. Oury and W. Swierstra. The power of Pi. In *Proc. of ICFP'08*, pages 39–50, 2008.

T. Schrijvers, S. L. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP'09*, pages 341–352, 2009.

T. Sheard and L. Fegaras. A fold for all seasons. In *Proc. of FPCA'93*, pages 233–242, 1993.

M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput*, 11(4):763–783, 1982.

A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. of FPCA'95*, pages 306–313, 1995.

N. Yoneda. On Ext and exact sequences. *Jour. Fac. Sci. Univ. Tokyo*, pages 507–576, 1960.