Universes for Generic Programs and Proofs in Dependent Type Theory

Marcin Benke, Peter Dybjer, and Patrik Jansson

Department of Computing Science,
Chalmers University of Technology,
412 96 Göteborg, Sweden
{marcin,peterd,patrikj}@cs.chalmers.se

Abstract. We show how to write generic programs and proofs in Martin-Löf type theory. To this end we consider several extensions of Martin-Löf's logical framework for dependent types. Each extension has a universes of codes (signatures) for inductively defined sets with generic formation, introduction, elimination, and equality rules. These extensions are modeled on Dybjer and Setzer's finitely axiomatized theories of inductive-recursive definitions, which also have a universe of codes for sets, and generic formation, introduction, elimination, and equality rules. However, here we consider several smaller universes of interest for generic programming and universal algebra. We thus formalize one-sorted and many-sorted term algebras, as well as iterated, parameterized, generalized, and indexed inductive definitions. We also show how to extend the techniques of generic programming to these universes. Most of the definitions in the paper have been implemented using the proof assistant Alfa for dependent type theory.

Introduction

The basic idea of generic functional programming is to define generic functions by induction on the definition of a data type. A simple example of a generic function is Boolean equality: indeed, a generic equality test is provided by languages such as SML (where it is built-in) and Haskell (where it is a derivable class). More powerful examples include generic map combinators, and generic iteration and recursion over inductive datatypes. Generic definitions are highly reusable (one definition can be used at many different instances) and adaptive (changing a datatype is as easy as changing a parameter), and they are therefore well suited for building libraries of programs, theorems and proofs. This research area has been explored under different names by Böhm & Berarducci [BB85] (universal algebra) by Backhouse et al. [B+91] (Squiggol), by Bird et al. [BdMH96] (generic functional programming), by Jay [Jay95,Jay01] (shape polymorphism), by Jansson & Jeuring [JJ97,Jan00] (polytypic programming) and by Hinze & Jeuring [HJ] (Generic Haskell).

A basic example of a dependent types is the type of vectors (lists) `Vect n`, which depends on the length `n` of the vector. With dependent types we can also capture more complex invariants of datastructures, for example, ordered lists, balanced trees, AVL-trees, red-black-trees, etc. We can in fact express more or less arbitrary properties of programs and data structures using dependent types.

Recently several authors [PR99,Ben01,AM02,Nor02] have noted that the techniques of generic programming can profitably be expressed in dependently typed languages such as Martin-Löf type theory, the Calculus of Constructions, and the programming language Cayenne [Aug98]. Combining dependent types with the idea of generic programming we can capture a class of datatypes as a universe — a set of codes and an interpretation function — and generic functions become functions over this universe (functions indexed by these codes).

In this paper we continue the programme of writing generic programs and proofs in dependent type theory initiated by Pfeifer and Rueß [PR99]. Like them we work in a total dependent type theory and use the Curry-Howard identification of propositions and types for representing logical notions. (Although they work in the impredicative Calculus of Constructions and we in Martin-Löf type theory, this difference is not essential in this context.) The main contributions of the present paper are the following:

– We introduce several universes of codes for inductively defined sets. One of these (parameterized term algebras) coincide with Pfeifer and Rueß' universe, but we also have universes for indexed inductive definitions (inductive families) and generalized inductive definitions, which have not been considered before in the context of generic programming.
– We make a link with the work on extending Martin-Löf type theory with general notions of inductive and inductive-recursive definitions. In particular we build on the work by Dybjer and Setzer [DS99,DS01a] who obtained finite axiomatizations of inductive-recursive definitions by introducing a universe of codes for such definitions. In this way we get generic elimination rules for inductively defined set which specialize to the standard elimination rules for particular sets in Martin-Löf type theory. Our generic elimination rules are different from the generic elimination rule used by Pfeifer and Rueß, and perhaps easier to use.
– We give generic proofs of reflexivity and substitutivity of Boolean equality, and thus continue the programme of demonstrating that it is possible in practice to carry out generic proofs of properties of functions defined on generic datatypes. (Pfeifer and Rueß already gave one example in their paper: a generic proof that constructors are injective.)
– We give a new approach to formalizing universal algebra in dependent type theory. We introduce universes for one- and many sorted term algebras, parameterized term algebras, and term algebras with infinitary operations.

Plan of the paper. We introduce the logical framework in section 1. Then we introduce several different universes corresponding to various interesting classes of inductive definitions. We begin in section 2 by introducing a universe of codes for *homogeneous term algebras*, that is, initial one-sorted algebras over a signature. After this each section deals with one particular extension of the simplest case.

Section 3 describes *iterated inductive definitions* of algebraic datatypes — one algebraic datatype can be used in the definition of another. Section 4 explores infinitary inductive definitions (such as the Brouwer ordinals). Section 5 discusses the codes for parameterized datatypes. Section 6 introduces *heterogeneous term algebras*, that is, initial many-sorted algebras. In Section 7 we introduce a universe for dependent datatypes (inductive families). Finally, in section 8 we briefly introduce Dybjer and Setzer's theory **IID** of indexed inductive definitions, and conjecture that the universes in Sections 3-7 are subuniverses of the universe of indexed inductive definitions, and that the respective theories are all subtheories of **IID**.

All the Alfa-code defining generic functions and universes in this paper is available from `www.cs.chalmers.se/~patrikj/poly/gendt/`.

1 The logical framework for dependent types

Here we introduce Martin-Löf's logical framework, extended with sum types and finite types. It contains rules for dependent function types $(x : A) \to B$, dependent product types $(x : A) \times B$, and sum types $A + B$. Lambda-abstraction is written $\lambda x.e$ as in lambda calculus and application is mainly written $f\,e$ but sometimes arguments are put in index position f_e. Pairing is written (d, e) and we use angle brackets for pairing of functions: $\langle f, g\rangle$. Injections are written $\mathrm{Inl}, \mathrm{Inr}$. We also have a case analysis construct for which we don't give explicit syntax; instead we write definition by cases using pattern matching equations. Moreover, we include the finite sets. We write $\mathrm{Fin}\, n$ for the finite set with n elements $0, \ldots, n-1$, but also $\mathbf{0} = \mathrm{Fin}\, 0$ for the empty set, and $\mathbf{1} = \mathrm{Fin}\, 1$ for the one-element set (whose unique element is denoted by $\star$). We write $[A]$ for the list type, with constructors $[\,]$ and $(::)$ for empty and non-empty lists, respectively. We also use various common notational conventions, including superscripts and and argument-hiding, to improve readability. As usual for logical frameworks, we assume β and η-equality for dependent function, dependent product, and unit types, but just β-equality for sums.

Moreover, we have the type Set containing sets in Martin-Löf's sense, that is, inductive data types defined by their constructors (introduction rules). Furthermore, for each set $A : \mathrm{Set}$, there is the type $\mathrm{El}\, A$ of its elements. We follow the usual convention and just write A for $\mathrm{El}\, A$ (as in universes à la Russell [ML84]). Set is also closed under dependent functions, dependent products, units, and sums. El commutes with all these constructions and we will therefore use the same notation for them on the set level as on the type level.

2 One-sorted term algebras

The simplest class of inductive types is the class of (carriers of) term algebras T_Σ for a one-sorted signature Σ. This is by no means the first formalization of one-sorted algebras in dependent type theory. But we include it here for pedagogical reasons and in order to show some interesting generic proofs in a setting where they are reasonably easy to grasp.

A one-sorted signature is nothing but a finite list of natural numbers, representing the arities of the operations of the signature. Examples are the empty type with $\Sigma = [\,]$, the natural numbers with $\Sigma = [0, 1]$, the Booleans with $\Sigma = [0, 0]$, lists of Booleans with $\Sigma = [0, 1, 1]$, and binary trees without information in the nodes with $\Sigma = [0, 2]$. Thus we introduce our first universe as the set of signatures $\mathrm{Sig} = [\mathrm{Nat}] : \mathrm{Set}$, and the decoding function $\mathrm{T} : \mathrm{Sig} \to \mathrm{Set}$, which maps a signature to (the carrier of) its term algebra.

We also include formation, introduction, (large) elimination, and equality rules for Nat and Sig.

2.1 Generic formation, introduction, elimination, and equality rules

These rules are best understood by recalling the initial algebra semantics of the term algebras T_Σ [GTW78]. Categorically, if F is an endofunctor (sometimes called the "pattern functor") on a category then an F-algebra with carrier X is an arrow

$$F\,X \xrightarrow{f} X$$

Initial algebra semantics of term algebras over a signature Σ states that the F_Σ-algebra

$$\mathrm{F}_\Sigma \mathrm{T}_\Sigma \xrightarrow{\mathrm{Intro}_\Sigma} \mathrm{T}_\Sigma$$

is initial among F_Σ-algebras, that is, for any other F_Σ-algebra

$$\mathrm{F}_\Sigma\, C \xrightarrow[d]{} C$$

there is a (unique) arrow $\mathrm{iter}_\Sigma\, C\, d$ which makes the following diagram commute.

$$\begin{tikzcd}
\mathrm{F}_\Sigma \mathrm{T}_\Sigma \arrow[r, "\mathrm{Intro}_\Sigma"] \arrow[d, "\mathrm{F}_\Sigma(\mathrm{iter}_\Sigma\, C\, d)"'] & \mathrm{T}_\Sigma \arrow[d, "\mathrm{iter}_\Sigma\, C\, d"] \\
\mathrm{F}_\Sigma\, C \arrow[r, "d"'] & C
\end{tikzcd}$$

The pattern functor F_Σ is a functor on a category of types. It has two parts, an object and an arrow part:

$$\begin{aligned}
&\mathrm{F}^0_\Sigma : \mathrm{Set} \to \mathrm{Set} \\
&\mathrm{F}^1_\Sigma : (A, B : \mathrm{Set}) \to (A \to B) \to \mathrm{F}^0_\Sigma A \to \mathrm{F}^0_\Sigma B
\end{aligned}$$

which are defined by induction on Σ : Sig. We will often suppress the superscripts 0 and 1 and use F both for the object and the arrow part. We will also often hide Set-arguments (in this case A and B). Informally,

$$\mathrm{F}_{[n_1,\ldots,n_m]}\, X = X^{n_1} + \cdots + X^{n_m}$$

Formally, we define the object part

$$\begin{aligned} \mathrm{F}^0{}_{[\,]}\, X &= \mathbf{0} \\ \mathrm{F}^0{}_{n::\Sigma}\, X &= X^n + \mathrm{F}^0_{\Sigma}\, X \end{aligned}$$

where $X^0 = \mathbf{1}$ and $X^{n+1} = X \times X^n$, and we define the arrow part

$$\begin{aligned} \mathrm{F}^1_{n::\Sigma}\, f\,(\mathrm{Inl}\, x) &= \mathrm{Inl}\,(f^n\, x) \\ \mathrm{F}^1_{n::\Sigma}\, f\,(\mathrm{Inr}\, x) &= \mathrm{Inr}\,(\mathrm{F}^1_{\Sigma}\, f\, x) \end{aligned}$$

where for $f : X \to Y$ we have

$$\begin{aligned} f^n \ &: \ X^n \to Y^n \\ f^0 \star &= \star \\ f^{n+1}\,(x, xs) &= (f\, x, f^n\, xs) \end{aligned}$$

Note that the base case $\mathrm{F}^1_{[\,]}$ is vacuous, since $\mathrm{F}^0_{[\,]}\, X = \mathbf{0}$.

Now we get generic rules of formation, introduction, and elimination for the set T_Σ for each Σ : Sig, by giving formal axioms expressing the existence of *weakly* initial F_Σ-algebras. As usual in type theory, inductively defined sets only have weak (β-like) rules. Full initiality would amount to have strong (η-like) rules as well. These rules are expressed as new constants which are added to the logical framework from Section 1:

$$\begin{aligned} \mathrm{T}_\Sigma &: \mathrm{Set} \\ \mathrm{Intro}_\Sigma &: \mathrm{F}^0_\Sigma \mathrm{T}_\Sigma \to \mathrm{T}_\Sigma \\ \mathrm{iter}_\Sigma &: (C : \mathrm{Set}) \to (\mathrm{F}^0_\Sigma C \to C) \to (\mathrm{T}_\Sigma \to C) \end{aligned}$$

Furthermore, there is the equality rule

$$\mathrm{iter}_\Sigma\, C\, d\,(\mathrm{Intro}_\Sigma\, x) = d\,(\mathrm{F}^1_\Sigma\,(\mathrm{iter}_\Sigma\, C\, d)\, x)$$

Note that this is iteration, rather than recursion, and that C is a set rather than a family of sets, as in typical type-theoretic rules. In the next subsection we define the corresponding recursor.

We can also use large elimination, so that C can be a large type, for example, the type Set of sets, but we do not write this rule down formally.

2.2 Generic induction schema

The elimination rule obtained directly from the above initial algebra diagram only captures definition by iteration. We would like a more general Martin-Löf style generic elimination rule, which captures proof by induction and definition by primitive (or structural) recursion. To do this we use the following instance of the initial algebra diagram (see Coquand & Paulin [CP90], Dybjer & Setzer [DS99,DS00]):

$$\begin{tikzcd}
\mathrm{F}_\Sigma \mathrm{T}_\Sigma \arrow[rr, "\mathrm{Intro}_\Sigma"] \arrow[d, "\mathrm{F}_\Sigma\langle id, \mathrm{rec}_\Sigma\, C\, d\rangle"'] \arrow[dr, "f"] & & \mathrm{T}_\Sigma \arrow[d, "\langle id, \mathrm{rec}_\Sigma\, C\, d\rangle"] \\
\mathrm{F}_\Sigma((x : \mathrm{T}_\Sigma) \times\, C\, x) \arrow[r, "\cong"'] & (y : \mathrm{F}_\Sigma\, \mathrm{T}_\Sigma) \times \mathrm{F}^{\mathrm{IH}}_\Sigma\, C\, y \arrow[r, "e"'] & (x : \mathrm{T}_\Sigma) \times C\, x
\end{tikzcd}$$

where

$$\begin{aligned} e(y, z) &= (\mathrm{Intro}_\Sigma\, y, d\, y\, z) \\ f y &= (y, \mathrm{F}^{\mathrm{map}}_\Sigma\, C\, (\mathrm{rec}_\Sigma\, C\, d)\, y) \end{aligned}$$

In order to get the usual shape of the elimination rule, we have introduced the auxiliary constructions

$$\begin{gathered} \mathrm{F}^{\mathrm{IH}}_\Sigma : (\mathrm{T}_\Sigma \to \mathrm{Set}) \to (\mathrm{F}_\Sigma\, \mathrm{T}_\Sigma \to \mathrm{Set}) \\ \mathrm{F}^{\mathrm{IH}}_{[n_0,\ldots,n_i]}\, C\, (\mathrm{In}_i\, (x_1, \ldots, x_{n_i})) = C\, x_1 \times \cdots \times C\, x_{n_i} \end{gathered}$$

and

$$\begin{gathered} \mathrm{F}^{\mathrm{map}}_\Sigma : (C : \mathrm{T}_\Sigma \to \mathrm{Set}) \to ((x : \mathrm{T}_\Sigma) \to C\, x) \to ((y : \mathrm{F}_\Sigma\, \mathrm{T}_\Sigma) \to \mathrm{F}^{\mathrm{IH}}_\Sigma\, C\, y) \\ \mathrm{F}^{\mathrm{map}}_{[n_0,\ldots,n_i]}\, C\, h\, (\mathrm{In}_i\, (x_1, \ldots, x_{n_i})) = (h\, x_1, \ldots, h\, x_n) \end{gathered}$$

as in Dybjer & Setzer.

Hence the elimination rule is

$$\mathrm{rec}_\Sigma : (C : \mathrm{T}_\Sigma \to \mathrm{Set}) \to ((y : \mathrm{F}_\Sigma\, \mathrm{T}_\Sigma) \to \mathrm{F}^{\mathrm{IH}}_\Sigma\, C\, y \to C\, (\mathrm{Intro}_\Sigma\, y)) \to (x : \mathrm{T}_\Sigma) \to C\, x$$

The equality rule is

$$\mathrm{rec}_\Sigma\, C\, d\, (\mathrm{Intro}_\Sigma\, y) = d\, y\, (\mathrm{F}^{\mathrm{map}}_\Sigma\, C\, (\mathrm{rec}_\Sigma\, C\, d)\, y)$$

As before we may use a large version of this elimination too, where C can be an arbitrary family of types, not just a family of sets.

2.3 Examples

Generic size. A special case of the initial algebra diagram. Let $\Sigma = [n_1, \ldots, n_m]$.

$$\begin{tikzcd}
\mathrm{T}^{n_1}_\Sigma + \cdots + \mathrm{T}^{n_m}_\Sigma \arrow[r, "\mathrm{Intro}_\Sigma"] \arrow[d, "\mathrm{size}^{n_1}_\Sigma + \cdots + \mathrm{size}^{n_m}_\Sigma"'] & \mathrm{T}_\Sigma \arrow[d, "\mathrm{size}_\Sigma"] \\
\mathrm{Nat}^{n_1} + \cdots + \mathrm{Nat}^{n_m} \arrow[r, "\mathrm{sizestep}_\Sigma"'] & \mathrm{Nat}
\end{tikzcd}$$

In our implementation, it becomes

$$\begin{aligned}
\mathrm{size}_\Sigma &= \mathrm{iter}_\Sigma\,\mathrm{sizestep}_\Sigma \\
\mathrm{sizestep}_{n::\Sigma}\;(\mathrm{Inl}\,xs) &= 1 + \mathrm{sum}_n\,xs \\
\mathrm{sizestep}_{n::\Sigma}\;(\mathrm{Inr}\,y) &= \mathrm{sizestep}_\Sigma\;y
\end{aligned}$$

where

$$\mathrm{sum} : (n : \mathrm{Nat}) \to \mathrm{Nat}^n \to \mathrm{Nat}$$

is a function summing the elements of a vector of natural numbers.

Generic destructor. We can also define the generic destructor

$$\mathrm{out}_\Sigma : \mathrm{T}_\Sigma \to \mathrm{F}_\Sigma\mathrm{T}_\Sigma$$

$$\mathrm{out}_\Sigma\,x = \mathrm{rec}_\Sigma\,(\lambda x.\mathrm{F}_\Sigma\mathrm{T}_\Sigma)\,(\lambda yz.y)$$

In effect, the destructor gives us pattern matching on Intro_Σ as we can see by specializing the equality rule for rec_Σ:

$$\mathrm{out}_\Sigma\,(\mathrm{Intro}_\Sigma\,x) = x$$

Generic equality.

$$\begin{tikzcd}
\mathrm{T}_\Sigma^{n_1} + \cdots + \mathrm{T}_\Sigma^{n_m} \arrow[r, "\mathrm{Intro}_\Sigma"] \arrow[d, "\mathrm{eq}_\Sigma^{n_1} + \cdots + \mathrm{eq}_\Sigma^{n_m}"'] & \mathrm{T}_\Sigma \arrow[d, "\mathrm{eq}_\Sigma"] \\
(\mathrm{T}_\Sigma \to \mathrm{Bool})^{n_1} + \cdots + (\mathrm{T}_\Sigma \to \mathrm{Bool})^{n_m} \arrow[r, "\mathrm{eqstep}_\Sigma"'] & \mathrm{T}_\Sigma \to \mathrm{Bool}
\end{tikzcd}$$

where informally

$$\mathrm{eqstep}_\Sigma\,(\mathrm{In}_i\,(p_1,\ldots,p_{n_i}))\,(\mathrm{Intro}\,(\mathrm{In}_i\,(y_1,\ldots,y_{n_i}))) = p_1\,y_1 \;\wedge\; \cdots \;\wedge\; p_{n_i}y_{n_i}$$

and for $i \neq j$

$$\mathrm{eqstep}_\Sigma\,(\mathrm{In}_i\,(p_1,\ldots,p_{n_i}))\,(\mathrm{Intro}\,(\mathrm{In}_j\,(y_1,\ldots,y_{n_j}))) = \mathrm{False}$$

Formally,

$$\begin{aligned}
&\mathrm{eqstep}_\Sigma : \mathrm{F}_\Sigma(\mathrm{T}_\Sigma \to \mathrm{Bool}) \to (\mathrm{T}_\Sigma \to \mathrm{Bool}) \\
&\mathrm{eqstep}_\Sigma \quad y\;x = \mathrm{eqstep'}_\Sigma\,\mathrm{T}_\Sigma\,y\,(\mathrm{out}_\Sigma\,x)
\end{aligned}$$

$$\begin{aligned}
\mathrm{eqstep'}_\Sigma \quad &: (Y : \mathrm{Set}) \to \mathrm{F}_\Sigma(Y \to \mathrm{Bool}) \to \mathrm{F}_\Sigma Y \to \mathrm{Bool} \\
\mathrm{eqstep'}_{n::\Sigma} \quad &Y\,(\mathrm{Inl}\,f_0)\;(\mathrm{Inl}\,x_0) = \mathrm{eqstep_ar}_n\,Y\,f_0\,x_0 \\
\mathrm{eqstep'}_{n::\Sigma} \quad &Y\,(\mathrm{Inl}\,f_0)\;(\mathrm{Inr}\,x_1) = \mathrm{False} \\
\mathrm{eqstep'}_{n::\Sigma} \quad &Y\,(\mathrm{Inr}\,f_1)\;(\mathrm{Inl}\,x_0) = \mathrm{False} \\
\mathrm{eqstep'}_{n::\Sigma} \quad &Y\,(\mathrm{Inr}\,f_1)\;(\mathrm{Inr}\,x_1) = \mathrm{eqstep'}_\Sigma\,Y\,f_1\,x_1
\end{aligned}$$

$$\begin{aligned}
\text{eqstep_ar}_n &: (Y : Set) \to (Y \to Bool)^n \to Y^n \to Bool \\
\text{eqstep_ar}_0 &\quad Y \star \star = \text{True} \\
\text{eqstep_ar}_{m+1} &\quad Y \langle f_1, f_2 \rangle \langle x_1, x_2 \rangle = (f_1\, x_1) \wedge \text{eqstep_ar}_m\, Y\, f_2\, x_2
\end{aligned}$$

The case of $\Sigma = [\,]$ is, again, vacuous.

2.4 Generic proof of reflexivity of equality

To state the reflexivity we need to convert booleans truth values to propositional truth values. This can also be seen as a universe construction — the booleans are codes for (just) the two types $\mathbf{0}$ and $\mathbf{1}$:

$$\begin{aligned}
|\cdot| &: \text{Bool} \to \text{Set} \\
|\text{False}| &= \mathbf{0} \\
|\text{True}| &= \mathbf{1}
\end{aligned}$$

Boolean "and" can be lifted to the type level: (only this one case is inhabited)

$$\begin{aligned}
&\text{liftAnd} : (a, b : \text{Bool}) \to |a| \to |b| \to |a \wedge b| \\
&\text{liftAnd} \quad \text{True True} \star \star = \star
\end{aligned}$$

When this lemma is used, the first two parameters will be omitted for brevity.

Now we define reflexivity and local reflexivity:

$$\begin{aligned}
\text{rel} &: \text{Set} \to \text{Set} \\
\text{rel} &\quad X = X \to X \to \text{Bool} \\
\text{reflexive} &: (X : \text{Set}) \to (r : \text{rel}\, X) \to Set \\
\text{reflexive} &\quad A\, r = (x : X) \to |r\, x\, x| \\
\text{lref} &: (X : \text{Set}) \to (r : \text{rel}\, X) \to (x : X) \to \text{Set} \\
\text{lref} &\quad X\, x = |r\, x\, x|
\end{aligned}$$

As the definition of equality used some auxiliary definitions, we begin by proving lemmas about properties of these definitions:

$$\begin{aligned}
&\text{ref_eqstep_ar}_n \quad : (X : \text{Set}) \to (e : \text{rel}\, X) \to (x : X^n) \to \\
&\qquad\qquad\qquad (\text{lref}\, e)^n\, x \to |\text{eqstep_ar}_n\, X\, (e^n\, x)\, x| \\
&\text{ref_eqstep_ar}_0 \quad X\, e \star \star = \star \\
&\text{ref_eqstep_ar}_{m+1}\, X\, e\, \langle x_1, x_2 \rangle\, \langle ih_1, ih_2 \rangle = \text{liftAnd}\, ih_1 (\text{ref_eqstep_ar}_m\, X\, e\, x_2\, ih_2)
\end{aligned}$$

$$\begin{aligned}
&\text{ref_eqstep}'_\Sigma \quad : (X : \text{Set}) \to (e : rel\, X) \to (x : F_\Sigma\, X) \to \\
&\qquad\qquad\quad \mathrm{F}^{\mathrm{IH}}_{\Sigma,X} (\text{lref}\, e)\, x \to |\text{eqstep}'_\Sigma\, X\, (F^1_\Sigma\, e\, x)\, x| \\
&\text{ref_eqstep}'_{n::\Sigma} \quad X\, e\, (\text{Inl}\, x_0) = \text{ref_eqstep_ar}_n\, X\, e\, x_0 \\
&\text{ref_eqstep}'_{n::\Sigma} \quad X\, e\, (\text{Inr}\, x_1) = \text{ref_eqstep}'_\Sigma\, X\, e\, x_1
\end{aligned}$$

$$\begin{aligned}
&\text{ref_eqstep}_\Sigma : (e : rel\, \mathrm{T}_\Sigma) \to (x : \mathrm{F}_\Sigma\, \mathrm{T}_\Sigma) \to \\
&\qquad\qquad \mathrm{F}^{\mathrm{IH}}_\Sigma (\text{lref}\, e)\, x \to |\text{eqstep}_\Sigma\, (F^1_\Sigma\, e\, x)\, (\text{Intro}\, x)| \\
&\text{ref_eqstep}_\Sigma \quad e = \text{ref_eqstep}'_\Sigma\, \mathrm{T}_\Sigma\, e
\end{aligned}$$

Finally, we wrap up the proof using the recursor:

$\text{ref\_eq}_\Sigma \;:\; \text{reflexive}(\text{eq}_\Sigma)$
$\text{ref\_eq}_\Sigma = \text{rec}_\Sigma(\text{lref}\,\text{eq}_\Sigma)\,(\text{ref\_eqstep}_\Sigma\,\text{eq}_\Sigma)$

The proof of substitutivity (that equal element can not be separated by any predicate) follows exactly the same pattern and can be found on this paper's homepage.

3 Iterated induction

The one-sorted term algebras provide a quite limited class of inductive datatypes for programming. A first generalization is to admit iterated induction, that is, in an introduction rule (typing rule for a constructor) we can refer to a previously defined datatype. For example, to define the set of lists of natural numbers ListNat, we refer to the set of natural numbers:

$$\begin{aligned} \text{Nil} &: \text{ListNat} \\ \text{Cons} &: \text{Nat} \to \text{ListNat} \to \text{ListNat} \end{aligned}$$

To obtain this class of iterated inductive definitions, we redefine the type of signatures

$$\text{Sig} = [\text{Arity}]$$

where an arity now is defined by the following inductive definition:

$$\begin{aligned} \text{Zero} &: \text{Arity} \\ \text{Rec} &: \text{Arity} \to \text{Arity} \\ \text{NonRec} &: \text{Sig} \to \text{Arity} \to \text{Arity} \end{aligned}$$

(As always, we include elimination and equality rules for arities and signatures here too.)

Note that for one-sorted term algebras, an arity was just a natural number, that is, essentially something generated by Zero and Rec. Here we have added a new constructor NonRec for a non-recursive argument of a constructor. (A non-recursive argument is often called a side-condition.) If NonRec is applied to a signature Σ it means that the non-recursive argument ranges over the previously defined type T_Σ.

For example, lists of natural numbers have a signature

$$\Sigma_{\text{ListNat}} = [\text{Zero}, \text{NonRec}\,\Sigma_{\text{Nat}}\,(\text{Rec}\,\text{Zero})]$$

where $\Sigma_{\text{Nat}} = [\text{Zero}, \text{Rec}\,\text{Zero}]$.

The generic type-theoretic rules for iterated induction are the same as before, except that we need to extend the definitions of the pattern functor to the case of NonRec:

$$\mathrm{F}_{[\alpha_1,\ldots,\alpha_n]}\,X = \mathrm{F}^{\text{ar}}_{\alpha_1}\,X + \ldots + \mathrm{F}^{\text{ar}}_{\alpha_n}\,X$$

$$\begin{aligned}
\mathrm{F}^{\mathrm{ar}}_{\mathrm{Nil}}\,X &= \mathbf{1}\\
\mathrm{F}^{\mathrm{ar}}_{\mathrm{Rec}\,\alpha}\,X &= X \times \mathrm{F}_\alpha\,X\\
\mathrm{F}^{\mathrm{ar}}_{\mathrm{NonRec}\,\Sigma\,\alpha}\,X &= \mathrm{T}_\Sigma \times \mathrm{F}_\alpha\,X
\end{aligned}$$

We can now define generic size and equality functions for all sets defined by the class of iterated inductive definitions given in this section.

The Alfa-code for iterated inductive definitions including definitions of size and equality is available on the paper's homepage.

Remark. Note that ListNat was the type of signatures for one-sorted algebras in the previous section. So having extended the notion of signatures we can define the family of term algebras T_Σ for $\Sigma : \mathrm{Sig}$ as an internal family in the extended theory. Even more, we can (maybe using extensional equality) derive the rules for one-sorted term algebras from the rules for iterated inductive definitions.

4 Generalized induction

So far we have considered ordinary (or finitary) inductive definitions, that is, we have only considered finite arities. We can consider a notion of one-sorted algebras which allows infinitary operations, by changing the notion of a signature from a list of natural numbers to a list of sets. (Grätzer's book "Universal Algebra" [Grä79] is in fact about universal algebras with infinitary operations, although working in classical set theory, his arities are possibly infinite ordinal numbers.)

So we let[1]

$$\mathrm{Sig} = [\mathrm{Set}] : \mathrm{Type}$$

and modify the pattern functor:

$$\mathtt{F}_{[I_1,\dots,I_m]}\,X = (I_1 \to X) + \cdots + (I_m \to X)$$

For example the signatures for the empty type, the unit type, natural numbers, and the Brouwer ordinals $\mathcal{O}$ can be expressed as follows

$$\begin{aligned}
\Sigma_{\mathbf{0}} &= [\,]\\
\Sigma_{\mathbf{1}} &= [\mathbf{0}]\\
\Sigma_{\mathrm{Nat}} &= [\mathbf{0}, \mathbf{1}]\\
\Sigma_{\mathcal{O}} &= [\mathbf{0}, \mathbf{1}, \mathrm{T}_{\Sigma_{\mathrm{Nat}}}]
\end{aligned}$$

The Brouwer ordinals are sometimes called the second number class. We can define the third number class by having an operation with arity $\mathcal{O}$, and so on for the all the higher number classes.

[1] since we allow the arities to be arbitrary elements of Set, the signatures are no longer elements of Set, but of the second universe: Type is a universe such that Set : Type.

We cannot define decidable equality over the class of generalized inductive definitions. However, we have the following generic definition of a propositional extensional equality:

$$\begin{aligned}
\mathrm{eq}_\Sigma\ &:\ \mathrm{T}_\Sigma \to \mathrm{T}_\Sigma \to \mathrm{Set}\\
\mathrm{eq}_\Sigma &= \mathrm{iter}_\Sigma\ \mathrm{eqstep}_\Sigma\\
\mathrm{eqstep}_\Sigma\ &:\ \mathrm{F}_\Sigma(\mathrm{T}_\Sigma \to \mathrm{Set}) \to \mathrm{T}_\Sigma \to \mathrm{Set}\\
\mathrm{eqstep}_\Sigma\ &\quad ps\ x = \mathrm{eqstep}'_\Sigma\,\mathrm{T}_\Sigma\, ps\,(\mathrm{out}_\Sigma\, x)
\end{aligned}$$

where

$$\mathrm{eqstep}'_\Sigma : (Y : \mathrm{Type}) \to \mathrm{F}_\Sigma\,(Y \to \mathrm{Set}) \to \mathrm{F}_\Sigma\, Y \to \mathrm{Set}$$

$$\begin{aligned}
\mathrm{eqstep}'_{I::\Sigma}\, Y\,(\mathrm{Inl}\, f)\,(\mathrm{Inl}\, x) &= (i : I) \to f\, i\,(x\, i)\\
\mathrm{eqstep}'_{I::\Sigma}\, Y\,(\mathrm{Inl}\, f)\,(\mathrm{Inr}\, y) &= \mathbf{0}\\
\mathrm{eqstep}'_{I::\Sigma}\, Y\,(\mathrm{Inr}\, g)\,(\mathrm{Inl}\, x) &= \mathbf{0}\\
\mathrm{eqstep}'_{I::\Sigma}\, Y\,(\mathrm{Inr}\, g)\,(\mathrm{Inr}\, y) &= \mathrm{eqstep}'_\Sigma\, Y\, g\, y
\end{aligned}$$

We can also combine generalized and iterated induction. In particular we can consider signatures where branchings range over previously defined inductive types, leading to the following notion of signature:

$$\mathrm{Sig} = [\mathrm{Sig}] : \mathrm{Set}$$

that is, arities and signatures are mutually defined. If we present this definition with constructors we get

$$\begin{aligned}
\mathrm{Nil} &: \mathrm{Sig}\\
\mathrm{Rec} &: \mathrm{Sig} \to \mathrm{Sig} \to \mathrm{Sig}
\end{aligned}$$

The pattern functor is (using list notation):

$$\mathtt{F}_{[\Sigma_1,\ldots,\Sigma_m]}\, X = (\mathrm{T}_{\Sigma_1} \to X) + \cdots + (\mathrm{T}_{\Sigma_m} \to X)$$

In this setting we have

$$\begin{aligned}
\Sigma_{\mathbf{0}} &= [\,]\\
\Sigma_{\mathbf{1}} &= [\Sigma_{\mathbf{0}}] = [\,[\,]\,]\\
\Sigma_{\mathrm{Nat}} &= [\Sigma_{\mathbf{0}}, \Sigma_{\mathbf{1}}] = [\,[\,]\,, [\,[\,]\,]\,]\\
\Sigma_{\mathcal{O}} &= [\Sigma_{\mathbf{0}}, \Sigma_{\mathbf{1}}, \Sigma_{\mathrm{Nat}}] = [\,[\,]\,, [\,[\,]\,], [\,[\,]\,, [\,[\,]\,]\,]\,]
\end{aligned}$$

5 Parameterized term algebras

So far we have only considered constant term algebras, that is, T_Σ is a constant set. However, many interesting generic functions range over parameterized types. We therefore extend our notion of signature to account for parameters, and the decoding function now takes a signature and returns a parameterized term algebra, that is, it is a function

$$\mathrm{T} : \mathrm{Sig} \to (\mathrm{Set} \to \mathrm{Set})$$

The universe of parameterized term algebras is the introduced by Pfeifer & Rueß [PR99], and if we also add iterated induction we obtain the case considered in Jansson & Jeuring [JJ97].

Parameterized term algebras are term algebras which depend on one or several parameter types. We consider here the case of one parameter for simplicity. Examples of parameterized term algebras are the type $[A]$ of lists of parameter type A with constructors

$$\begin{aligned} \mathrm{Nil} &: (A : \mathrm{Set}) \to [A] \\ \mathrm{Cons} &: (A : \mathrm{Set}) \to A \to [A] \to [A] \end{aligned}$$

We therefore add a new constructor, Par, for arities[2] (compared with the homogeneous case)

$$\begin{aligned} \mathrm{Nil} &: \mathrm{Arity} \\ \mathrm{Rec} &: \mathrm{Arity} \to \mathrm{Arity} \\ \mathrm{Par} &: \mathrm{Arity} \to \mathrm{Arity} \end{aligned}$$

The signature for parametric lists $[A]$ is then

$$[\mathrm{Zero}, \mathrm{Par}\,(\mathrm{Rec}\,\mathrm{Zero})]$$

The initial algebra diagram for iteration now needs to take parameters into account:

$$\begin{tikzcd} \mathrm{F}_\Sigma\,A\,(\mathrm{T}_\Sigma\,A) \arrow[r, "\mathrm{Intro}_\Sigma\,A"] \arrow[d, "\mathrm{F}_\Sigma\,A\,(\mathrm{iter}_\Sigma\,A\,C\,d)"'] & \mathrm{T}_\Sigma\,A \arrow[d, "\mathrm{iter}_\Sigma\,A\,C\,d"] \\ \mathrm{F}_\Sigma\,A\,C \arrow[r, "d"'] & C \end{tikzcd}$$

We need to extend the definition of the pattern functor to the case of parameters:

$$\mathrm{F}^{\mathrm{ar}}_{\mathrm{Par}\,\alpha}\,A\,X = A \times \mathrm{F}^{\mathrm{ar}}_{\alpha}\,A\,X$$

[2] This gives us a notion of unary parameterized term algebra; it is straightforward to generalize this to n-ary parameterized algebras by instead having

$$\mathrm{Par} : (n : \mathrm{Nat}) \to \mathrm{Fin}\,n \to \mathrm{Arity} \to \mathrm{Arity}$$

The diagram for induction:

$$\begin{tikzcd}
\mathrm{F}_\Sigma\, A\, (\mathrm{T}_\Sigma\, A) \arrow[rr, "\mathrm{Intro}_\Sigma"] \arrow[d, "\mathrm{F}_\Sigma\, A\, \langle 1, \mathrm{rec}_\Sigma\, A\, C\, d \rangle"'] \arrow[dr] & & \mathrm{T}_\Sigma\, A \arrow[d, "\langle 1, \mathrm{rec}_\Sigma\, A\, C\, d \rangle"] \\
\mathrm{F}_\Sigma\, A\, ((x : \mathrm{T}_\Sigma) \times C\, x) \arrow[r, "\cong"] & (y : \mathrm{F}_\Sigma\, A\, (\mathrm{T}_\Sigma\, A)) \times \mathrm{F}^{\mathrm{IH}}_\Sigma\, A\, C\, y \arrow[r] & (x : \mathrm{T}_\Sigma) \times C\, x
\end{tikzcd}$$

As already mentioned, parameterized term algebras, are almost as powerful as the universe used in PolyP [JJ97]. In fact, it is sufficiently close to PolyP that the majority of the polytypic library functions [JJ98] carry over immediately.

When we consider a universe with parameterized types, many natural generic definitions share a common pattern: they lift a function from the parameter level to the parameterized type level. To show this pattern we introduce a few type synonyms and use these in the type signatures for (generic) size, equality, map and zip. (Here $A, B, C : \mathrm{Set}$ and $\Sigma : \mathrm{Sig}$.)

$$\begin{array}{lll}
\mathrm{Size} & A & = A \to \mathrm{Nat} \\
\mathrm{Eq} & A & = A \to A \to \mathrm{Bool} \\
\mathrm{Map} & A\ B & = A \to B \\
\mathrm{Zip} & A\ B\ C & = A \to B \to \mathrm{Maybe}\ C
\end{array}$$

The set $\mathrm{Maybe}\, A$ has constructors $\mathrm{Nothing} : \mathrm{Maybe}\, A$ and $\mathrm{Just} : A \to \mathrm{Maybe}\, A$ for each $A : \mathrm{Set}$.

$$\begin{array}{rlllll}
\mathrm{sizeBy}_\Sigma : & (A : \mathrm{Set}) & \to & \mathrm{Size}\ \ A & \to & \mathrm{Size}\ \ (\mathrm{T}_\Sigma\, A) \\
\mathrm{eqBy}_\Sigma : & (A : \mathrm{Set}) & \to & \mathrm{Eq}\ \ A & \to & \mathrm{Eq}\ \ (\mathrm{T}_\Sigma\, A) \\
\mathrm{map}_\Sigma : & (A, B : \mathrm{Set}) & \to & \mathrm{Map}\ A\ B & \to & \mathrm{Map}\ (\mathrm{T}_\Sigma\, A)\ (\mathrm{T}_\Sigma\, B) \\
\mathrm{zipWith}_\Sigma : & (A, B, C : \mathrm{Set}) & \to & \mathrm{Zip}\ \ A\ B\ C & \to & \mathrm{Zip}\ \ (\mathrm{T}_\Sigma\, A)\ (\mathrm{T}_\Sigma\, B)\ (\mathrm{T}_\Sigma\, C)
\end{array}$$

All these functions are straightforward to implement over this universe, see the code on the paper's homepage.

The application $\mathrm{zipWith}_\Sigma\ op\ x\, y$ compares x and y and succeeds iff they share the same structure, and all element comparisons (using op) succeed. The result shares the same structure as x and y and contains the results from the successful applications of op. With $\mathrm{Bool} \simeq \mathrm{Maybe}\, \mathbf{1}$ the equality test eqBy_Σ can be seen as a special case of $\mathrm{zipWith}_\Sigma$:

$$\begin{aligned}
\mathrm{Eq}\, A = A \to A \to \mathrm{Bool} &\simeq A \to A \to \mathrm{Maybe}\, \mathbf{1} = \mathrm{Zip}\, A\, A\, \mathbf{1} \\
\mathrm{eqBy}_\Sigma\, A\ op\ x\, y &= \mathrm{forget}_{\mathrm{T}_\Sigma \mathbf{1}}\, (\mathrm{zipWith}_\Sigma\, A\, A\, \mathbf{1}\ op\ x\, y)
\end{aligned}$$

where $\mathrm{forget} : (X : \mathit{Set}) \to \mathrm{Maybe}\ X \to \mathrm{Bool}$. The familiar zip function from generic functional programming is also an instance:

$$\mathrm{zip}_\Sigma\, A\, B = \mathrm{zipWith}_\Sigma\, A\, B\, (A, B)\, (\lambda x\, y.\mathrm{Just}\, (x, y))$$

6 Many-sorted term algebras

We shall now consider many-sorted term algebras, giving rise to a simple class of mutually inductive definitions, see also Capretta [Cap99]. This is the main class of term algebras considered in algebraic specification theory, following the work by the ADJ-group [GTW78].

For simplicity we consider many-sorted algebras with finitely many sorts, and no parameters. It is easy to add parameters. Note also that the iterated inductive definitions in section 3 are subsumed by the mutual inductive definitions here.

The type of signatures for n-sorted algebras is now

$$\mathrm{Sig}_n = \mathrm{Fin}\, n \to [\mathrm{Arity}_n], \quad \text{where} \quad \mathrm{Arity}_n = [\mathrm{Fin}\, n]$$

That is, a signature consists of n lists of arities, one for each sort. An arity is a list of numbers $< n$, denoting the sorts of the arguments of an operation.

As a simple example, consider the following mutual definition of the even and odd numbers:

$$\begin{aligned} \mathrm{SuccEven} &: \mathrm{Even} \to \mathrm{Odd} \\ \mathrm{Zero} &: \mathrm{Even} \\ \mathrm{SuccOdd} &: \mathrm{Odd} \to \mathrm{Even} \end{aligned}$$

The many-sorted signature is

$$\begin{aligned} \Sigma\, 0 &= [[1]] \\ \Sigma\, 1 &= [[\,], [0]] \end{aligned}$$

Other examples include trees and forests. More generally, abstract syntax trees for context-free grammars are also many-sorted algebras. The diagram for initial n-sorted algebras is

$$\begin{tikzcd} \mathrm{F}_\Sigma\, \mathrm{T}_\Sigma\, i \arrow[r, "\mathrm{Intro}_\Sigma\, i"] \arrow[d, "\mathrm{F}_\Sigma\, (\mathrm{iter}_\Sigma\, d\, i)"'] & \mathrm{T}_\Sigma\, i \arrow[d, "\mathrm{iter}_\Sigma\, d\, i"] \\ \mathrm{F}_\Sigma\, C\, i \arrow[r, "d\, i"'] & C\, i \end{tikzcd}$$

where $i : \mathrm{Fin}\, n$.

We do not have room either for displaying the diagram for the full elimination (induction) rule, nor for the definition of generic size and equality. However, as many-sorted algebras can be viewed as a special case of inductive families, appropriate definitions can be obtained by specializing the definitions found in Section 7.

7 Finitary indexed induction

In this section we consider a general class of finitary indexed inductive definitions. To define a family indexed by some set I, we first define the universe Sig_I:

$$\begin{aligned}\mathrm{Nil} &: \mathrm{Sig}_I \\ \mathrm{NonRec} &: (A : \mathrm{Set}) \to (A \to \mathrm{Sig}_I) \to \mathrm{Sig}_I \\ \mathrm{Rec} &: I \to \mathrm{Sig}_I \to \mathrm{Sig}_I\end{aligned}$$

Here Nil represents the base case — an inductive definition with no premise; NonRec represents the non-recursive case — adding a side condition $a : A$; and Rec represents the recursive case — adding a recursive premise. The pattern functor F_γ is defined by induction on $\gamma : \mathrm{Sig}\, I$:

$$\begin{aligned}\mathrm{F}_\gamma \;&:\; (I \to \mathrm{Set}) \to \mathrm{Set} \\ \mathrm{F}_{\mathrm{Nil}}\, X &= \mathbf{1} \\ \mathrm{F}_{\mathrm{NonRec}\, A\, \phi}\, X &= (x : A) \times \mathrm{F}_{\phi\, x}\, X \\ \mathrm{F}_{\mathrm{Rec}\, i\, \Sigma}\, X &= (X\, i) \times \mathrm{F}_\Sigma\, X\end{aligned}$$

Further, for every $\Sigma : I \to \mathrm{Sig}_I$ we have

$$\begin{aligned}\mathrm{T}_{I,\Sigma} &: I \to \mathrm{Set} \\ \mathrm{Intro}_{I,\Sigma} &: (i : I) \to \mathrm{F}_{\Sigma\, i}\, \mathrm{T}_{I,\Sigma} \to \mathrm{T}_{I,\Sigma}\, i\end{aligned}$$

There is also an elimination and an equality rule. Note that many-sorted algebras are a special case. We get n-sorted algebras if $I = \mathrm{Fin}\, n$, if arities are only built up by Zero and Rec, and where the only use of NonRec is for building up lists of arities.

The initial algebra diagram looks the same as in the many-sorted case. The type-theoretic rules for are:

$$\begin{aligned}\mathrm{T}_\Sigma &: I \to \mathrm{Set} \\ \mathrm{Intro}_\Sigma &: (i : I) \to \mathrm{F}_\Sigma \mathrm{T}_\Sigma i \to \mathrm{T}_\Sigma i \\ \mathrm{iter}_\Sigma &: (C : I \to \mathrm{Set}) \to ((i : I) \to \mathrm{F}_\Sigma\, \mathrm{T}_\Sigma\, i \to C\, i) \to (i : I) \to \mathrm{T}_\Sigma\, i \to C\, i \\ \mathrm{rec}_\Sigma &: (C : (i : I) \to \mathrm{T}_{I,\Sigma}\, i \to \mathrm{Type}) \\ &\quad \to ((i : I) \to (y : \mathrm{F}_{\Sigma\, i}\, \mathrm{T}_{I,\Sigma}) \to \mathrm{F}^{\mathrm{IH}}_{\Sigma\, i}\, \mathrm{T}_{I,\Sigma}\, C\, y \to C\, i\, (\mathrm{Intro}_\Sigma\, i\, y)) \\ &\quad \to (i : I) \to (x : \mathrm{T}_{I,\Sigma}\, i) \to C\, i\, x\end{aligned}$$

There is also equality rules that we do not display here.

An example of an inductive family (Agda-style) is the family of binary search trees, indexed by pairs of natural numbers (the lower and upper bound):

$$\mathrm{BST} : \mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Set}$$

The introduction rules are

$$\begin{aligned}
&\mathrm{C}_0 : (lb, ub : \mathrm{Nat}) \to (lb < ub) \to \mathrm{BST}\,(lb, ub)\\
&\mathrm{C}_1 : (lb, ub, root : \mathrm{Nat}) \to (lb < root) \to (root < ub) \to\\
&\qquad\qquad \to \mathrm{BST}\,(lb, root) \to \mathrm{BST}\,(root, ub) \to \mathrm{BST}\,(lb, ub)
\end{aligned}$$

Written as "arities" they become

$$\begin{aligned}
\mathrm{arity}\,(lb, ub)\,0 &= \mathrm{NonRec}\,(lb < ub)\,(\lambda p.\mathrm{Nil})\\
\mathrm{arity}\,(lb, ub)\,1 &= \mathrm{NonRec}\,\mathrm{Nat}\\
&\qquad (\lambda root.\mathrm{NonRec}(lb < root)(\lambda h.\mathrm{NonRec}(root < ub)\\
&\qquad (\lambda h'.(\mathrm{Rec}\,(lb, r)(\mathrm{Rec}\,(r, ub)\ \mathrm{Nil})))))
\end{aligned}$$

Thus the signature for the family BST becomes

$$\Sigma\,(lb, ub) = \mathrm{NonRec}\,(\mathrm{Fin}\,2)\,(\mathrm{arity}\,(lb, ub))$$

We can now write generic size function over this universe

$$\mathrm{size}_{I,\Sigma} : (i : I) \to \mathrm{T}_{I,\Sigma}\, i \to \mathrm{Nat}$$

However, to define equality

$$\mathrm{eq}_{I,\Sigma} : (i : I) \to \mathrm{T}_{I,\Sigma}\, i \to \mathrm{T}_{I,\Sigma}\, i \to \mathrm{Bool}$$

we need to restrict NonRec by allowing it to range only over sets with decidable equality (so called datoids):

$$\mathrm{NonRec} : (D : \mathrm{Datoid}) \to (|D| \to \mathrm{Sig}_I) \to \mathrm{Sig}_I$$

where $|D|$ is the carrier of the datoid D.

We have also added parameters to our universe for finitary indexed inductive definitions and thus equipped written $\mathrm{zipWith}_{I,\Sigma}$ from section 5. We refer to the Alfa-implementation for details.

8 The theory of indexed inductive definitions

In each of sections 3-7 we have presented a universe consisting of a set Sig of signatures and a family of sets T_Σ for each Σ : Sig. The formation, introduction, elimination, and equality rules for Sig (and in some cases for Arity) and the generic introduction, elimination, and equality rules for T_Σ defines an extension of the logical framework from section 2. Thus in each section we define a version of Martin-Löf type theory with a different collection of inductive definitions. In each of these theories we can write generic programs and proofs by induction on the signature. The idea is to choose a universe of signatures which is appropriate for a particular application.

However, each time we change universe we also change theory. This is of course unsatisfactory - we would like to be able to do generic programming over different universes in *one* theory. So we would like to have an all encompassing theory which can swallow all of the previous theories. For this purpose we could use the the theory of indexed inductive-recursive definitions $\mathbf{IIR}^{ext}$ (with extensional equality) given by Dybjer and Setzer [DS01a]. In this theory all of our universes can be defined but to actually work out these embeddings in detail is a task outside the scope of this paper.

In fact, since induction-recursion does not play a role in this paper, it would suffice with the theory of indexed inductive definitions **IID** (with extensionality). This is a natural upper bound of the theories presented in section 3-7. We present it here briefly.

IID has a universe OP_I of codes for sets, where I is a previously constructed set. The pattern functor F_γ is defined by induction on OP_I where OP_I is inductively defined by the following constructors:

$$\begin{aligned}
\mathrm{Nil} &: \mathrm{OP}_I \\
\mathrm{NonRec} &: (A : \mathrm{Set}) \to (A \to \mathrm{OP}_I) \to \mathrm{OP}_I \\
\mathrm{Rec} &: (A : \mathrm{Set}) \to (A \to I) \to \mathrm{OP}_I \to \mathrm{OP}_I
\end{aligned}$$

Here Nil represents the base case - an inductive definition with no premise; NonRec represents the non-recursive case - adding a side condition $a : A$; and Rec represents the recursive case - adding a recursive premise. (Note that we have generalized inductive definitions, that is, there is an A-indexed family of recursive premises. This is the only difference between **IID** and the theory of finitary indexed inductive definitions in the previous section.)

F_γ is defined by induction on $\gamma : \mathrm{OP}_I$, where

$$\begin{aligned}
\mathrm{F}_{\mathrm{Nil}}\, X &= \mathbf{1} \\
\mathrm{F}_{\mathrm{NonRec}\, A\, \phi}\, X &= (x : A) \times \mathrm{F}_{\phi\, x}\, X \\
\mathrm{F}_{\mathrm{Rec}\, A\, i\, \gamma}\, X &= ((x : A) \to X\,(i\, x)) \times \mathrm{F}_\gamma\, X
\end{aligned}$$

We refer to [DS01a,DS01b] for a full explanation of the theory **IIR** (and thus implicitly of **IID**).

IID is a suitable general framework for generic programming, since we conjecture that the theories in Sections 3-7 are definable in **IID** in the following senses. (We have however not yet given a a rigorous proof of this conjecture.) Firstly, the sets Sig of signatures for one-sorted algebras (possibly with iterated induction) has a code in $\mathrm{OP}_{\mathbf{1}}$ and is thus definable in **IID**. Moreover, each code in Sig can be mapped to a code in $\mathrm{OP}_{\mathbf{1}}$ and the decoding function can be obtained by composing the decoding function for $\mathrm{OP}_{\mathbf{1}}$ with this map. Furthermore the signature Sig for parameterized term algebras also has a code in $\mathrm{OP}_{\mathbf{1}}$. Here a code in Sig can be mapped to a function $\mathrm{Set} \to \mathrm{OP}_{\mathbf{1}}$, and the decoding for Sig can again be obtained by composing the decoding function for $\mathrm{OP}_{\mathbf{1}}$ with this map. The family Sig_n is also definable in **IID** and each code in $\mathrm{Sig}\, n$ can be mapped to a code in $\mathrm{OP}_{\mathrm{Fin}\, n}$, and again decoding of $\mathrm{Sig}\, n$ can be obtained from decoding of $\mathrm{OP}_{\mathrm{Fin}\, n}$. The situation with generalized induction in section 7 is similar to the situation with one-sorted algebras, except that as it stands Sig is here a "large" inductive definition, that is, it is a type rather than a set. This size problem can be solved if we replace the current large inductive definitions with an analogous small one. There is a similar size problem which prevents us from defining OP_I in section 8 as a set in **IID**.

9 Related work

PolyP and Generic Haskell PolyP [JJ97] as in "polytypic" (= generic) programming, is an extension of Haskell. Polytypic functions are defined by induction on a universe of codes for "regular datastructures" (roughly the universe of our section 5).

In Generic Haskell [HJ] (the successor of PolyP) the universe is generalized to include mutually recursive, higher order kinded and nested datatypes. This allows the full class of Haskell datatypes to be expressed but also restricts the set of definable generic functions.

Many datatypes with invariants can be simulated in Haskell using nested and higher-order kinded datatypes, but these types can be more directly expressed using dependent types.

Combining dependent types and generic programming. The research on this topic goes in two different directions. On the one hand Altenkirch and McBride [AM02] and Norell [Nor02] show how to encode Generic Haskell-style programming using dependent types. Here the setting is that of general recursive functional programming where the class of recursive datatypes includes for example nested datatypes.

On the other hand the work of Pfeifer and Rueß [PR99] and Benke [Ben01] are about extending the technique of generic programming to "total" type theories such as the Calculus of Construction and the Alfa proof assistant respectively. The idea here is to stay within a logical system based on the Curry-Howard isomorphism and therefore the type system ensures that all programs terminate by only allowing restricted forms of recursion. In this setting we can both write generic programs and write generic proofs of properties of those programs. In fact, experiments of one of the authors [Ben02] show that generic proofs of equality properties, such as equivalence, decidability and substitutivity can be actually simpler than the corresponding non-generic proofs.

The present paper continues the programme set out by Pfeifer and Rueß. Firstly, we introduce several universes of codes for inductive datatypes of interest for generic programming and universal algebra. One of them is Pfeifer and Rueß' universe parameterized term algebras. Others include universes for infinitary (generalized) inductive types and inductively defined families, neither of which have been considered for generic programming before. Furthermore, Pfeifer and Rueß only had one generic proof about a datatype: a proof that constructors are injective. Here we give some more examples: proofs of reflexivity and substitutivity of generic equality. As the reader will see, these proofs are non-trivial! To facilitate generic proofs we provide an elimination constant which captures primitive recursion rather than iteration (as in Pfeifer and Rueß).

Inductive definitions in dependent type theory We also connect work on inductive definitions in type theory with work on generic programming. Although the papers by Dybjer and Setzer [DS99,DS00,DS01b] contain related ideas, and in particular give generic formation, introduction, elimination, and equality rules for inductive-recursive definitions, they do not discuss the connection with practical generic programming – the generic programs and proofs in the paper have metatheoretic rather than practical interest. Furthermore, for the purpose of practical generic programming the universe of inductive-recursive definitions is too large. This is the reason why we introduce several smaller subuniverses of inductive types.

Universal algebra in dependent type theory. Bayley [Bay98] and Ruys [Ruy99] formalized one-sorted term algebras in dependent type theory. Capretta [Cap99] proposed several ways to formalizing many-sorted term algebras, including using Petersson-Synek trees [PS89] and extending dependent type theory with so called recursive families of inductive types.

References

[AM02] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In J. Gibbons and J. Jeuring, editors, *Pre-Proc. WCGP'02*, 2002. (Final proc. to be published by Kluwer Acad. Publ.).

[Aug98] L. Augustsson. Cayenne — a language with dependent types. In *Proc. ICFP'98*. ACM Press, September 1998.

[B+91] R.C. Backhouse et al. Relational catamorphisms. In B. Möller, editor, *Constructing Programs from Specifications*, pages 287–318. North-Holland, 1991.

[Bay98] A. Bayley. *The Machine-Checked Literate Formalisation of Algebra in Type Theory*. PhD thesis, University of Manchester, 1998.

[BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BdMH96] R. Bird, O. de Moor, and P. Hoogendijk. Generic functional programming with types and relations. *J. of Func. Prog.*, 6(1):1–28, 1996.

[Ben01] M. Benke. Some tools for computer-assisted theorem proving in Martin-Löf type theory. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs — Supplemental Proceedings*. University of Edinburgh, 2001.

[Ben02] M. Benke. Towards generic programming in type theory. Presentation at Annual ESPRIT BRA TYPES Meeting, Berg en Dal. Submitted for publication, available from `http://www.cs.chalmers.se/~marcin/Papers/Notes/nijmegen.ps.gz`, April 2002.

[Cap99] V. Capretta. Universal algebra in type theory. In Y. Bertot et al., editors, *Proc. TPHOLs '99*, volume 1690 of *LNCS*, pages 131–148. Springer-Verlag, 1999.

[Cap02] V. Capretta. *Abstraction and Computation*. PhD thesis, Univ. of Nijmegen, the Netherlands, April 2002.

[CP90] T. Coquand and C. Paulin. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[DS99] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *Proc. TLCA'99*, volume 1581 of *LNCS*, pages 129–146. Springer-Verlag, Berlin, 1999.

[DS00] P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 2000. To appear.

[DS01a] P. Dybjer and A. Setzer. Indexed induction-recursion. In R. Kahle et al, editor, *Proof Theory in Computer Science*, volume 2183 of *LNCS*, pages 93–113. Springer Verlag, October 2001.

[DS01b] P. Dybjer and A. Setzer. Indexed induction-recursion. long version, submitted for publication, available from `http://www.cs.chalmers.se/~peterd/`, 2001.

[Grä79] G. Grätzer. *Universal Algebra*. Springer-Verlag, second edition, 1979.

[GTW78] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice-Hall, 1978.

[HJ] R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. To appear in the lecture notes of the Summer School on Generic Programming, LNCS Springer-Verlag, 2002/2003.

[Jan00] P. Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers Univ. of Tech. and Göteborg Univ., Sweden, May 2000.

[Jay95] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[Jay01] C.B. Jay. Distinguishing data structures and functions: The constructor calculus and functorial types. In S. Abramsky, editor, *Proc. TLCA'01*, volume 2044 of *LNCS*, pages 217–239. Springer-Verlag, Berlin, 2001.

[JJ97] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *Proc. POPL'97*, pages 470–482. ACM Press, 1997.

[JJ98] P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[Nor02] 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`.

[PR99] H. Pfeifer and H. Rueß. Polytypic proof construction. In Y. Bertot, editor, *Proc. TPHOLs'99*, volume 1690 of *LNCS*, pages 55–72. Springer-Verlag, 1999.

[PS89] K. Petersson and D. Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Category Theory and Computer Science*, pages 128–140. Springer-Verlag, LNCS 389, 1989.

[Ruy99] M. Ruys. *Studies in Mechanical Verification of Mathematical Proofs*. PhD thesis, Katholieke Universiteit Nijmegen, 1999.