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 universe 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 universes of codes for sets, and generic formation, introduction, elimination, and equality rules. Here we consider several smaller universes of interest for generic programming and universal algebra. We formalize one-sorted and many-sorted term algebras, as well as iterated, generalized, parameterized, and indexed inductive definitions. We also show how to extend the techniques of generic programming to these universes. Furthermore, we give generic proofs of reflexivity and substitutivity of a generic equality test. 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 [HJ03] (Generic Haskell).

A basic example of a dependent type is the type of vectors (lists) $\mathrm{Vect}\, n$, which depends on the length n of the vector. With dependent types we can also

capture more complex invariants of datastructures, for example, balanced trees, binary search trees, AVL-trees, etc. Furthermore, using the Curry-Howard identification of propositions and sets, we can in fact express more or less arbitrary properties of programs and data structures in dependent type theory.

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 initiated by Pfeifer and Rueß [PR99] of writing generic programs and proofs in dependent type theory. 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) coincides with Pfeifer and Rueß' universe, but we also have universes for indexed inductive definitions (inductive families) and generalized (infinitary) 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,DS01] 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 sets 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 closer to the usual elimination rules for inductively defined sets.
- 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. In section 2 we introduce several different universes corresponding to various interesting classes of inductive definitions. We begin in 2.1 by introducing a universe of signatures for *homogeneous term algebras*, that is, initial one-sorted algebras

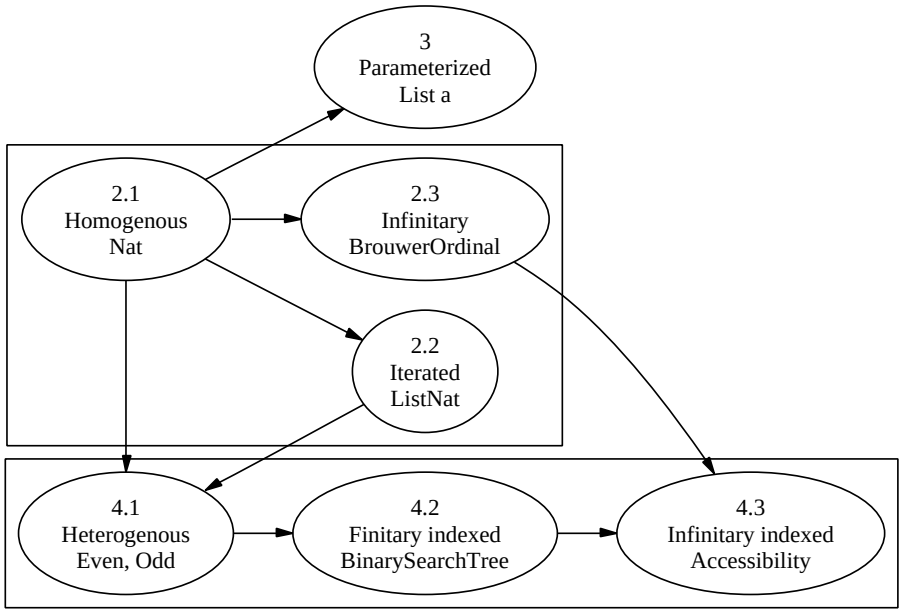


Fig. 1. Sections, universes and representative example types.

over a signature. We also show some generic programs and proofs for one-sorted term algebras. After this each section deals with one particular extension of the simplest case. Fig. 1 shows the relations between the different universes (an arrow from A to B means that A can be embedded in B). Subsection 2.2 describes *iterated inductive definitions* of algebraic datatypes — one algebraic datatype can be used in the definition of another. Subsection 2.3 explores generalized (infinitary) inductive definitions (such as the Brouwer ordinals). Section 3 discusses the codes for parameterized datatypes. Much generic programming is concerned with parameterized datatypes and we give several examples. Section 4 discusses several notions of indexed inductive definitions (inductive families). In 4.1 we present our coding of heterogenous term algebras. In 4.2 we introduce a universe for finitary indexed inductive definitions. In 4.3 we introduce Dybjer and Setzer's theory **IID** of generalized (infinitary) indexed inductive definitions. We conjecture that the universes in Sections 2.1–4.2 are subuniverses of the universe of indexed inductive definitions. Finally, in Section 5 we summarize related work.

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

Acknowledgement. The authors wish to thank an anonymous referee for many useful suggestions concerning the presentation of the paper.

1 The logical framework for dependent types

Martin-Löf's logical framework contains inference rules for deriving judgments of the following four forms: A Type, $A = A', a : A, a = a' : A$. Among these rules there are rules for dependent function types $(x : A) \to B$, the type Set of sets, and the types El A of elements for each A : Set.

Here we extend this framework with dependent product types $(x : A) \times B$, and the finite types $\mathbf{0}$, $\mathbf{1}$, and $\mathbf{2}$. As usual for logical frameworks, we assume β and η-equality for dependent function and dependent product. However, we only have β-rules for the finite types.

The type Set contains sets in Martin-Löf's sense, that is, inductive data types defined by their constructors (introduction rules). We follow the usual convention and just write A for El A (as in universes à la Russell [ML84]). Set is also closed under dependent functions, dependent products, and contains (codes for) $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{2}$. 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.

For a complete description of (essentially) the same logical framework, we refer to the appendix of Dybjer & Setzer [DS03a].

There are no rules for building sets (inductive datatypes) such as the set of natural numbers, sets of lists, vectors, trees, etc included in the logical framework. This is instead the purpose of the following sections: to give formal rules for constructing several different classes of such sets.

Convenient notation. We drop the type in the fourth form of judgment and abbreviate $a = a' : A$ by $a = a'$. Lambda-abstraction is written $\lambda x.e$ as in lambda calculus à la Curry. Application is mainly written fe but sometimes arguments are put in index position f_e. Pairing is written (d, e) with projections fst and snd. The sum type $A_0 + A_1$ is implemented as $(i : \mathbf{2}) \times A_i$ but injections are written Inl, Inr. We have a case analysis construct for which we don't give explicit syntax; instead we write definition by cases using pattern matching equations.

We write Fin n for the finite type with n elements denoted by $0, \ldots, n-1$. Formally, Fin $0 = \mathbf{1}$ and Fin $(m+1) = \mathbf{1} + \text{Fin}\, m$. In informal code we use n-ary sum types and we write In_i for the i-th injection.

We use angle brackets for pairing of functions: $\langle f, g \rangle$ is the function which returns the pair $(f\, x, g\, x)$ given the argument x. We also use various common notational conventions, including superscripts and and argument-hiding, to improve readability.

Although natural numbers, lists and vectors are not part of the logical framework, we already introduce some notational conventions for them which will be used later. We use Nat for natural numbers. We write $[A]$ for the list type, with constructors $[\,]$ and $(::)$ for empty and non-empty lists, respectively. We write X^n for the type of vectors of length n implemented as $X^0 = \mathbf{1}$ and $X^{n+1} = X \times X^n$. The informal notation for an element in X^n is $(x_1, \ldots, x_n)$. As a special case the unique element in $\mathbf{1}$ is denoted by $()$.

We can lift a function $f : X \to Y$ to operate on vectors:

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

When motivating the axioms of the different theories in Sections 2–4 we will draw several category-theoretic diagrams. These diagrams should be understood informally; the formal axioms are expressed purely type-theoretically. To aid the reader in seeing the correspondence between the informal diagrams and the formal axioms, we will sometimes keep redundant parentheses in type expressions, that is, we will sometimes write $A \to (B \to C)$ rather than the usual $A \to B \to C$.

2 Inductive definitions

2.1 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 $\Sigma_{\mathbf{0}} = [\,]$, the natural numbers with $\Sigma_{\mathrm{Nat}} = [0, 1]$, the Booleans with $\Sigma_{\mathrm{Bool}} = [0, 0]$, and binary trees without information in the nodes with $\Sigma_{\mathrm{Bin}} = [0, 2]$. Lists of Booleans has $\Sigma_{\mathrm{ListBool}} = [0, 1, 1]$, since it is generated by one constant for the empty list and one Cons for each Boolean:

$$\begin{aligned} \mathrm{NilBool} &: \mathrm{ListBool} \\ \mathrm{ConsTrue} &: \mathrm{ListBool} \to \mathrm{ListBool} \\ \mathrm{ConsFalse} &: \mathrm{ListBool} \to \mathrm{ListBool} \end{aligned}$$

Note however that we cannot code ListNat in this way, because we would then need infinitely many constructors.

Formally we introduce our first universe as the type of signatures $\mathrm{Sig} = [\mathrm{Arity}] = [\mathrm{Nat}]$, and the decoding function $\mathrm{T} : \mathrm{Sig} \to \mathrm{Set}$, which maps a signature to (the carrier of) its term algebra. In this first universe we also include formation, introduction, (large) elimination, and equality rules for Nat and Sig.

Generic formation, introduction, elimination, and equality rules. These rules are best understood by recalling initial algebra semantics of 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

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

Let F_Σ be the pattern functor associated with a signature Σ. Initial algebra semantics of the term algebra T_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.

$$\begin{array}{ccc} \mathrm{F}_\Sigma \mathrm{T}_\Sigma & \xrightarrow{\mathrm{Intro}_\Sigma} & \mathrm{T}_\Sigma \\ \Big\downarrow{\scriptstyle \mathrm{F}_\Sigma(\mathrm{iter}_\Sigma\, C\, d)} & & \Big\downarrow{\scriptstyle \mathrm{iter}_\Sigma\, C\, d} \\ \mathrm{F}_\Sigma\, C & \xrightarrow[d]{} & C \end{array}$$

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

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

which are defined by induction on $\Sigma : \mathrm{Sig}$. 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 X and Y). For example, the left vertical arrow $\mathrm{F}_\Sigma(\mathrm{iter}_\Sigma\, C\, d)$ in the commuting diagram above is an abbreviation of $\mathrm{F}^1_\Sigma\, \mathrm{T}_\Sigma\, C\, (\mathrm{iter}_\Sigma\, C\, d)$.

Informally, we define

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

The formal definition of F^0_Σ can be found in Fig. 2 and the formal definition of F^1_Σ is the following:

$$\begin{aligned} &\mathrm{F}^1 : (\Sigma : \mathrm{Sig}) \to (X, Y : \mathrm{Set}) \to (X \to Y) \to (\mathrm{F}^0_\Sigma\, X \to \mathrm{F}^0_\Sigma\, Y) \\ &\mathrm{F}^1_{n::\Sigma}\, X\, Y\, f\, (\mathrm{Inl}\, xs) = \mathrm{Inl}\, (f^n\, xs) \\ &\mathrm{F}^1_{n::\Sigma}\, X\, Y\, f\, (\mathrm{Inr}\, y) \;\; = \mathrm{Inr}\, (\mathrm{F}^1_\Sigma\, X\, Y\, f\, y) \end{aligned}$$

Note that the base case $\mathrm{F}^1_{[\,]}$ is vacuous, since $\mathrm{F}^0_{[\,]}\, X = \mathbf{0}$. In general, when we define a function by pattern matching, if the domain is empty for a certain combination of arguments, we don't write out that case.

Now we get generic rules for the set T_Σ for each $\Sigma : \mathrm{Sig}$, by giving formal axioms expressing the existence of *weakly* initial F_Σ-algebras. As usual in Martin-Löf type theory, inductively defined sets only have weak (β-like) rules. Full initiality would amount to having strong (η-like) rules as well.

The formation, introduction, and (simplified) elimination rule for T_Σ are expressed as the following three typings of new constants which are added to the logical framework from Section 1:

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

The generic equality rule is

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

We call the function argument d to the iterator the *step function* because it takes care of one step of the calculation with iter tying the recursive knot.

Note that the simplified elimination rule iter_Σ captures iteration, rather than primitive recursion, and that C is a set rather than a family of sets, as in typical type-theoretic rules. The full elimination rule rec_Σ is defined later in this subsection. Fig. 2 describes in detail the axioms and rules which together with the logical framework describes the theory of homogeneous algebras. 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.

Instances for natural numbers. Here we use the more compact notation $\underline{\mathrm{Nat}} = [0, 1]$ for the code for Nat and we note that $\mathrm{T}_{\underline{\mathrm{Nat}}} = \mathrm{Nat}$.

$$\begin{aligned} \mathrm{F}^0_{\underline{\mathrm{Nat}}}\, X &= \mathbf{1} + (X \times \mathbf{1} + \mathbf{0}) \cong \mathbf{1} + X \\ \mathrm{Intro}_{\underline{\mathrm{Nat}}} &: \mathbf{1} + (\mathrm{Nat} \times \mathbf{1} + \mathbf{0}) \to \mathrm{Nat} \cong \mathbf{1} + \mathrm{Nat} \to \mathrm{Nat} \\ \mathrm{iter}_{\underline{\mathrm{Nat}}} &: (C : \mathrm{Set}) \to (\mathbf{1} + (C \times \mathbf{1} + \mathbf{0}) \to C) \to (\mathrm{Nat} \to C) \\ &\cong (C : \mathrm{Set}) \to (\mathbf{1} + C \to C) \to (\mathrm{Nat} \to C) \\ &\cong (C : \mathrm{Set}) \to (C \times (C \to C)) \to (\mathrm{Nat} \to C) \end{aligned}$$

As the type of the step function is isomorphic to $C \times (C \to C)$ we are in effect supplying the iterator with one value for the base case and one function to iterate. The usual natural number constructors Zero and Succ can be expressed as follows:

$$\begin{aligned} \mathrm{Zero} &= \mathrm{Intro}_{\underline{\mathrm{Nat}}}\, (\mathrm{Inl}\, ()) \\ \mathrm{Succ}\, n &= \mathrm{Intro}_{\underline{\mathrm{Nat}}}\, (\mathrm{Inr}\, (\mathrm{Inl}\, (n, ()))) \end{aligned}$$

If this theory would be used as a base for practical generic programming, then the system would automatically extract the code for a datatype and recover the usual constructors from the definition of the datatype.

Examples of generic functions. We define a generic size function and a generic equality function. Formally, the generic definitions should be expressed using the elimination rules for arities and signatures (see Fig. 3), but in this presentation we use pattern matching and explicit recursion for readability.

$$
\begin{array}{ll}
\text{Arity} \;\; \text{Type} & \text{Sig} \;\; \text{Type} \\
\text{Zero} \;\; : \text{Arity} & [\,] \;\; : \text{Sig} \\
\text{Succ} \;\; : \text{Arity} \to \text{Arity} & (::) \;\; : \text{Arity} \to \text{Sig} \to \text{Sig}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{F}^{\mathrm{ar0}} : \text{Arity} \to \text{Set} \to \text{Set} & \mathrm{F}^{0} : \text{Sig} \to \text{Set} \to \text{Set} \\
\mathrm{F}^{\mathrm{ar0}}_{\mathrm{Zero}} \quad X = \mathbf{1} & \mathrm{F}^{0}_{[\,]} \quad X = \mathbf{0} \\
\mathrm{F}^{\mathrm{ar0}}_{\mathrm{Succ}\, m}\; X = X \times \mathrm{F}^{\mathrm{ar0}}_{m}\, X & \mathrm{F}^{0}_{n::\Sigma}\; X = \mathrm{F}^{\mathrm{ar0}}_{n}\, X + \mathrm{F}^{0}_{\Sigma}\, X
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{F}^{\mathrm{arIH}} : (n : \text{Arity}) \to (X : \text{Set}) \to & \mathrm{F}^{\mathrm{IH}} : (\Sigma : \text{Sig}) \to (X : \text{Set}) \to \\
\qquad (X \to \text{Set}) \to \mathrm{F}^{\mathrm{ar0}}_{n}\, X \to \text{Set} & \qquad (X \to \text{Set}) \to \mathrm{F}^{0}_{\Sigma}\, X \to \text{Set} \\
\mathrm{F}^{\mathrm{arIH}}_{\mathrm{Zero}} \quad X\, C\, () \qquad = \mathbf{1} & \mathrm{F}^{\mathrm{IH}}_{n::\Sigma}\, X\, C\, (\mathrm{Inl}\, xs) = \mathrm{F}^{\mathrm{arIH}}_{n}\, X\, C\, xs \\
\mathrm{F}^{\mathrm{arIH}}_{\mathrm{Succ}\, m}\; X\, C\, (x, xs) = C\, x \times \mathrm{F}^{\mathrm{arIH}}_{m}\, X\, C\, xs & \mathrm{F}^{\mathrm{IH}}_{n::\Sigma}\, X\, C\, (\mathrm{Inr}\, y) \;\; = \mathrm{F}^{\mathrm{IH}}_{\Sigma}\, X\, C\, y
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{F}^{\mathrm{armap}} : \; (n : \text{Arity}) \to (X : \text{Set}) \to & \mathrm{F}^{\mathrm{map}} : \; (\Sigma : \text{Sig}) \to (X : \text{Set}) \to \\
\qquad (C : X \to \text{Set}) \to & \qquad (C : X \to \text{Set}) \to \\
\qquad ((x : X) \to C\, x) \to & \qquad ((x : X) \to C\, x) \to \\
\qquad (xs : \mathrm{F}^{\mathrm{ar0}}_{n}\, X) \to \mathrm{F}^{\mathrm{arIH}}_{n}\, X\, C\, xs & \qquad (y : \mathrm{F}^{0}_{\Sigma}\, X) \to \mathrm{F}^{\mathrm{IH}}_{\Sigma}\, X\, C\, y \\
\mathrm{F}^{\mathrm{armap}}_{\mathrm{Zero}}\, X\, C\, f\, () \qquad = () & \mathrm{F}^{\mathrm{map}}_{n::\Sigma}\, X\, C\, f\, (\mathrm{Inl}\, xs) = \mathrm{F}^{\mathrm{armap}}_{n}\, X\, C\, f\, xs \\
\mathrm{F}^{\mathrm{armap}}_{\mathrm{Succ}\, m}\, X\, C\, f\, (x, xs) = (f\, x, \mathrm{F}^{\mathrm{armap}}_{m}\, X\, C\, f\, xs) & \mathrm{F}^{\mathrm{map}}_{n::\Sigma}\, X\, C\, f\, (\mathrm{Inr}\, y) \;\; = \mathrm{F}^{\mathrm{map}}_{\Sigma}\, X\, C\, f\, y
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{T} \quad : \text{Sig} \to \text{Set} & \\
\mathrm{Intro} : (\Sigma : \text{Sig}) \to \mathrm{F}^{0}_{\Sigma}\, \mathrm{T}_{\Sigma} \to \mathrm{T}_{\Sigma} & \\
\mathrm{rec} \quad : (\Sigma : \text{Sig}) \to (C : \mathrm{T}_{\Sigma} \to \text{Set}) \to & \\
\qquad ((y : \mathrm{F}^{0}_{\Sigma}\, \mathrm{T}_{\Sigma}) \to \mathrm{F}^{\mathrm{IH}}_{\Sigma}\, \mathrm{T}_{\Sigma}\, C\, y \to C\, (\mathrm{Intro}_{\Sigma}\, y)) \to & \\
\qquad (x : \mathrm{T}_{\Sigma}) \to C\, x & \\
\mathrm{rec}_{\Sigma}\, C\, d\, (\mathrm{Intro}_{\Sigma}\, y) = d\, y\, (\mathrm{F}^{\mathrm{map}}_{\Sigma}\, \mathrm{T}_{\Sigma}\, C\, (\mathrm{rec}_{\Sigma}\, C\, d)\, y) &
\end{array}
$$

Fig. 2. Axioms for the theory of homogeneous term algebras (large elimination rules can be added)

$$
\begin{aligned}
&\mathrm{rec}_{\mathrm{Arity}} : (C : \mathrm{Arity} \to \mathrm{Set}) \to C\,\mathrm{Zero} \to \\
&\qquad ((m : \mathrm{Arity}) \to C\,m \to C\,(\mathrm{Succ}\,m)) \to \\
&\qquad (n : \mathrm{Arity}) \to C\,n \\
&\mathrm{rec}_{\mathrm{Arity}}\,C\,z\,s\,\mathrm{Zero} \quad\;\; = z \\
&\mathrm{rec}_{\mathrm{Arity}}\,C\,z\,s\,(\mathrm{Succ}\,m) = s\,m\,(\mathrm{rec}_{\mathrm{Arity}}\,C\,z\,s\,m)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{rec}_{\mathrm{Sig}} : (C : \mathrm{Sig} \to \mathrm{Set}) \to C\,[\,] \to \\
&\qquad ((m : \mathrm{Arity}) \to (ms : \mathrm{Sig}) \to C\,ms \to C\,(m :: ms)) \to \\
&\qquad (ns : \mathrm{Sig}) \to C\,ns \\
&\mathrm{rec}_{\mathrm{Sig}}\,C\,n\,c\,[\,] \quad\;\; = n \\
&\mathrm{rec}_{\mathrm{Sig}}\,C\,n\,c\,(m :: ms) = c\,m\,ms\,(\mathrm{rec}_{\mathrm{Sig}}\,C\,n\,c\,ms)
\end{aligned}
$$

Fig. 3. Elimination rules for arities and signatures. (Again, large elimination rules can be added.)

Generic size. This is obtained as a special case of the initial algebra diagram. Let $\Sigma = [n_1, \ldots, n_m]$.

$$
\begin{array}{ccc}
\mathrm{T}_\Sigma^{n_1} + \cdots + \mathrm{T}_\Sigma^{n_m} & \xrightarrow{\mathrm{Intro}_\Sigma} & \mathrm{T}_\Sigma \\
\Big\downarrow{\scriptstyle \mathrm{size}_\Sigma^{n_1} + \cdots + \mathrm{size}_\Sigma^{n_m}} & & \Big\downarrow{\scriptstyle \mathrm{size}_\Sigma} \\
\mathrm{Nat}^{n_1} + \cdots + \mathrm{Nat}^{n_m} & \xrightarrow[\mathrm{sizestep}_\Sigma]{} & \mathrm{Nat}
\end{array}
$$

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.

For the special case of $\Sigma = \underline{\mathrm{Nat}}$ the step function simplifies to

$$
\begin{aligned}
&\mathrm{sizestep}_{\underline{\mathrm{Nat}}} : \mathbf{1} + (\mathrm{Nat} \times \mathbf{1} + \mathbf{0}) \quad\;\; \to \mathrm{Nat} \\
&\mathrm{sizestep}_{\underline{\mathrm{Nat}}}\;\; (\mathrm{Inl}\,()) \qquad\qquad\qquad\qquad\;\; = 1 \\
&\mathrm{sizestep}_{\underline{\mathrm{Nat}}}\;\; (\mathrm{Inr}\,(\mathrm{Inl}\,(\mathit{subsum}, ()))) = 1 + \mathit{subsum}
\end{aligned}
$$

Note that this means that $\mathrm{size}\,n = n + 1$ because the generic size counts the total number of Intro constructors in n (in this case both Zero and Succ).

Generic equality. A function for testing equality between two values naturally has two arguments, while the initial algebra diagram describes functions of one argument. Fortunately, the result type can be instantiated freely, so by returning a function we can easily simulate a two-argument function. It helps the reading of the types below to think about equality as a one-argument function returning a *recognizer* — a predicate which yields true only for values matching its internal value. The step function then receives a value containing recognizers for the substructures, and returns a recognizer for the top level. We obtain this diagram:

$$\begin{array}{ccc} \mathrm{T}_\Sigma^{n_1} + \cdots + \mathrm{T}_\Sigma^{n_m} & \xrightarrow{\quad\mathrm{Intro}_\Sigma\quad} & \mathrm{T}_\Sigma \\ \Big\downarrow{\scriptstyle \mathrm{eq}_\Sigma^{n_1} + \cdots + \mathrm{eq}_\Sigma^{n_m}} & & \Big\downarrow{\scriptstyle \mathrm{eq}_\Sigma} \\ (\mathrm{T}_\Sigma \to \mathrm{Bool})^{n_1} + \cdots + (\mathrm{T}_\Sigma \to \mathrm{Bool})^{n_m} & \xrightarrow[\mathrm{eqstep}_\Sigma]{} & (\mathrm{T}_\Sigma \to \mathrm{Bool}) \end{array}$$

where informally (let $\Sigma = [n_1, \ldots, n_m]$):

$$\begin{aligned} &\mathrm{eqstep}_\Sigma\, x\, (\mathrm{Intro}\, y) = \mathrm{recog\_all}_\Sigma\, \mathrm{T}_\Sigma\, x\, y \\ \\ &\mathrm{recog\_all}_\Sigma\, X : (X \to \mathrm{Bool})^{n_1} + \cdots + (X \to \mathrm{Bool})^{n_m} \to \\ &\qquad\qquad X^{n_1} + \cdots + X^{n_m} \to \mathrm{Bool} \\ &\mathrm{recog\_all}_\Sigma\, (\mathrm{In}_i\, (p_1, \ldots, p_{n_i}))\, (\mathrm{In}_i\, (y_1, \ldots, y_{n_i}))) = p_1\, y_1 \wedge \cdots \wedge p_{n_i}\, y_{n_i} \\ &\mathrm{recog\_all}_\Sigma\, (\mathrm{In}_i\, (p_1, \ldots, p_{n_i}))\, (\mathrm{In}_j\, (y_1, \ldots, y_{n_j}))) = \mathrm{False} \qquad \text{if } i \neq j \end{aligned}$$

Formally;

$$\begin{aligned} &\mathrm{eq}_\Sigma\ :\ \mathrm{T}_\Sigma \to (\mathrm{T}_\Sigma \to \mathrm{Bool}) \\ &\mathrm{eq}_\Sigma = \mathrm{iter}_\Sigma\ \mathrm{eqstep}_\Sigma \end{aligned}$$

$$\begin{aligned} &\mathrm{eqstep}_\Sigma : \mathrm{F}_\Sigma\, (\mathrm{T}_\Sigma \to \mathrm{Bool}) \to (\mathrm{T}_\Sigma \to \mathrm{Bool}) \\ &\mathrm{eqstep}_\Sigma\, x\, t = \mathrm{recog\_all}_\Sigma\, \mathrm{T}_\Sigma\, x\, (\mathrm{out}_\Sigma\, t) \end{aligned}$$

where $\mathrm{out}_\Sigma : \mathrm{T}_\Sigma \to \mathrm{F}_\Sigma \mathrm{T}_\Sigma$ is defined later in this subsection.

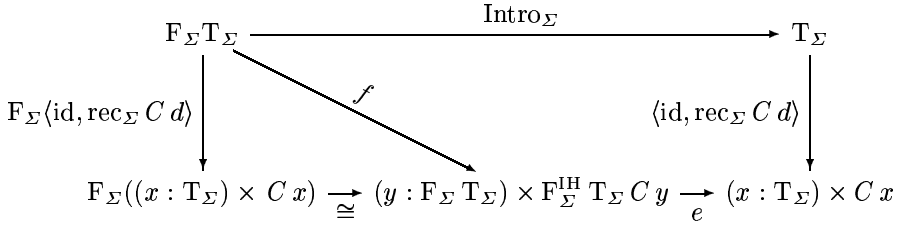
$$\begin{aligned} &\mathrm{recog\_all}_\Sigma : (X : \mathrm{Set}) \to \mathrm{F}_\Sigma\, (X \to \mathrm{Bool}) \to \mathrm{F}_\Sigma\, X \to \mathrm{Bool} \\ &\mathrm{recog\_all}_{n::\Sigma}\ \ X\ \ (\mathrm{Inl}\, fs)\ \ (\mathrm{Inl}\, xs) = \mathrm{and\_args}_n\, X\, fs\, xs \\ &\mathrm{recog\_all}_{n::\Sigma}\ \ X\ \ (\mathrm{Inr}\, x)\ \ (\mathrm{Inr}\, y)\ \ = \mathrm{recog\_all}_\Sigma\, X\, x\, y \\ &\mathrm{recog\_all}_{n::\Sigma}\ \ X\ \ (\mathrm{Inl}\, fs)\ \ (\mathrm{Inr}\, y)\ \ = \mathrm{False} \\ &\mathrm{recog\_all}_{n::\Sigma}\ \ X\ \ (\mathrm{Inr}\, x)\ \ (\mathrm{Inl}\, xs) = \mathrm{False} \end{aligned}$$

$$\begin{aligned} &\mathrm{and\_args}_n : (X : \mathrm{Set}) \to (X \to \mathrm{Bool})^n \to X^n \to \mathrm{Bool} \\ &\mathrm{and\_args}_0\ \ X\ \ ()\ \ \ \ ()\ \ \ \ = \mathrm{True} \\ &\mathrm{and\_args}_{m+1}\ \ X\ \ (p, ps)\ \ (x, xs) = p\, x \wedge \mathrm{and\_args}_m\, X\, ps\, xs \end{aligned}$$

If we instead work in the theory of parameterized term algebras (defined in Section 3), the whole definition of $\mathrm{recog\_all}_\Sigma$ could be replaced with $\mathrm{eqBy}_\Sigma\, (\lambda p.\lambda x.p\, x)$.

Generic induction schema. The elimination rule obtained directly from the initial algebra diagram earlier in this subsection 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 consider the following instance of the initial algebra diagram. Similar constructions can be found in Coquand & Paulin [CP90] and Dybjer & Setzer [DS99,DS03b]. We believe they are essential in practice for doing generic proofs.



where

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

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

$$\mathrm{F}^{\mathrm{IH}}_\Sigma : (X : \mathrm{Set}) \to (X \to \mathrm{Set}) \to (\mathrm{F}_\Sigma\, X \to \mathrm{Set})$$

$$\mathrm{F}^{\mathrm{IH}}_{[n_0,\ldots,n_m]}\, X\, C\, (\mathrm{In}_i\, (x_1, \ldots, x_{n_i})) = C\, x_1 \times \cdots \times C\, x_{n_i}$$

and

$$\begin{aligned} \mathrm{F}^{\mathrm{map}}_\Sigma : {} & (X : \mathrm{Set}) \to (C : X \to \mathrm{Set}) \to \\ & ((x : X) \to C\, x) \to \\ & ((y : \mathrm{F}_\Sigma\, X) \to \mathrm{F}^{\mathrm{IH}}_\Sigma\, X\, C\, y) \end{aligned}$$

$$\mathrm{F}^{\mathrm{map}}_{[n_0,\ldots,n_m]}\, X\, C\, f\, (\mathrm{In}_i\, (x_1, \ldots, x_{n_i})) = (f\, x_1, \ldots, f\, x_{n_i})$$

as in Dybjer & Setzer. Their formal definitions can be found in Fig. 2.

Hence the elimination rule is

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

The equality rule is

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

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.

Iteration is a special case of recursion. The diagram above only commutes up to extensional equality; we do not expect to derive the rules for rec_Σ from the rules for iter_Σ up to definitional equality, so we add the rules for rec_Σ as primitives. Conversely, we can however define iter_Σ by instantiating rec_Σ with a constant family $\lambda x.C$ and by ignoring the first argument to the step function. Thus the full elimination rule simplifies to the rule for iteration:

$$\mathrm{iter}_\Sigma\, C\, d\, e = \mathrm{rec}_\Sigma\, (\lambda x.C)\, (\lambda y.d)\, e$$

From this we can derive the equality rule for iter_Σ up to definitional equality.

Instances for natural numbers. We first instantiate F^{IH} to the code for Nat:

$$\begin{aligned}
&\mathrm{F}^{\mathrm{IH}}_{\underline{\mathrm{Nat}}} : (X : \mathrm{Set}) \to (X \to \mathrm{Set}) \to (\mathrm{F}_{\underline{\mathrm{Nat}}}\, X \to \mathrm{Set}) \\
&\mathrm{F}^{\mathrm{IH}}_{\underline{\mathrm{Nat}}}\, X\, C\, (\mathrm{Inl}\, ()) = \mathbf{1} \\
&\mathrm{F}^{\mathrm{IH}}_{\underline{\mathrm{Nat}}}\, X\, C\, (\mathrm{Inr}\, (\mathrm{Inl}\, (x, ()))) = C\, x
\end{aligned}$$

Then, by simplifying the type of the step function in $\mathrm{rec}_{\underline{\mathrm{Nat}}}$, we see that the step function contains the familiar base case and induction step from induction on natural numbers:

$$\begin{aligned}
&(y : \mathbf{1} + \mathrm{Nat}) \to \mathrm{F}^{\mathrm{IH}}_{\underline{\mathrm{Nat}}}\, \mathrm{Nat}\, C\, y \to C\, (\mathrm{Intro}_{\underline{\mathrm{Nat}}}\, y) \\
\cong\ &\text{By case analysis} \\
&((u : \mathbf{1}) \to \mathrm{F}^{\mathrm{IH}}_{\underline{\mathrm{Nat}}}\, \mathrm{Nat}\, C\, (\mathrm{Inl}\, u) \to C\, (\mathrm{Intro}_{\underline{\mathrm{Nat}}}\, (\mathrm{Inl}\, u))) \times \\
&((n : \mathrm{Nat}) \to \mathrm{F}^{\mathrm{IH}}_{\underline{\mathrm{Nat}}}\, \mathrm{Nat}\, C\, (\mathrm{Inr}\, n) \to C\, (\mathrm{Intro}_{\underline{\mathrm{Nat}}}\, (\mathrm{Inr}\, n))) \\
\cong\ &\text{Use definition of } \mathrm{F}^{\mathrm{IH}} \\
&(\mathbf{1} \to \mathbf{1} \to C\, (\mathrm{Intro}_{\underline{\mathrm{Nat}}}\, (\mathrm{Inl}\, ()))) \times \\
&((n : \mathrm{Nat}) \to C\, n \to C\, (\mathrm{Intro}_{\underline{\mathrm{Nat}}}\, (\mathrm{Inr}\, n))) \\
\cong\ &\text{Simplify and use definitions of Zero and Succ} \\
&C\, \mathrm{Zero} \times ((n : \mathrm{Nat}) \to C\, n \to C\, (\mathrm{Succ}\, n))
\end{aligned}$$

Thus the type of $\mathrm{rec}_{\underline{\mathrm{Nat}}}$ is (isomorphic to) the following:

$$\begin{aligned}
\mathrm{rec}_{\underline{\mathrm{Nat}}} :\ &(C : \mathrm{Nat} \to \mathrm{Set}) \to C\, \mathrm{Zero} \to \\
&((n : \mathrm{Nat}) \to C\, n \to C\, (\mathrm{Succ}\, n)) \to \\
&((x : \mathrm{Nat}) \to C\, x)
\end{aligned}$$

Note that this is exactly the type of the elimination rule for arities in Fig. 3.

Generic destructor. As a simple example of using rec we can 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 proof of reflexivity of equality To state the reflexivity we need to convert Boolean 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 $\mathbf{1}$:

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

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

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

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

We first define two convenient abbreviations:

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

where the first argument to lref will be hidden for brevity. The proof structure follows the same structure as the definition of equality. The top level proof is defined using the recursor:

$$\begin{aligned} &\mathrm{ref\_eq}_{\Sigma} \ : \ (t : \mathrm{T}_{\Sigma}) \to |\mathrm{eq}_{\Sigma}\, t\, t| \\ &\mathrm{ref\_eq}_{\Sigma} = \mathrm{rec}_{\Sigma}\, (\mathrm{lref}\, \mathrm{eq}_{\Sigma})\, (\mathrm{ref\_cons}_{\Sigma}\, \mathrm{T}_{\Sigma}\, \mathrm{eq}_{\Sigma}) \end{aligned}$$

The next step, ref_cons, discriminates between the constructors:

$$\begin{aligned} &\mathrm{ref\_cons}_{\Sigma} : (X : \mathrm{Set}) \to (e : \mathrm{rel}\, X) \to \\ &\qquad (x : \mathrm{F}_{\Sigma}\, X) \to (\mathrm{F}^{\mathrm{IH}}_{\Sigma}\, X\, (\mathrm{lref}\, e)\, x \to |\mathrm{recog\_all}_{\Sigma}\, X\, (\mathrm{F}^{1}_{\Sigma}\, e\, x)\, x|) \\ &\mathrm{ref\_cons}_{n::\Sigma}\ X\ e\ (\mathrm{Inl}\, xs) \quad = \mathrm{ref\_args}_{n}\, X\, e\, xs \\ &\mathrm{ref\_cons}_{n::\Sigma}\ X\ e\ (\mathrm{Inr}\, y) \quad = \mathrm{ref\_cons}_{\Sigma}\, X\, e\, y \end{aligned}$$

Finally, ref_args handles the arguments to the constructor:

$$\begin{aligned} &\mathrm{ref\_args}_{n} : (X : \mathrm{Set}) \to (e : \mathrm{rel}\, X) \to \\ &\qquad (xs : X^{n}) \to (\mathrm{lref}\, e)^{n}\, xs \to |\mathrm{and\_args}_{n}\, X\, (e^{n}\, xs)\, xs| \\ &\mathrm{ref\_args}_{0} \quad X\ e\ () \qquad () \qquad = () \\ &\mathrm{ref\_args}_{m+1}\ X\ e\ (x, xs) \qquad (ih, ihs) \quad = \mathrm{liftAnd}\, ih\, (\mathrm{ref\_args}_{m}\, X\, e\, xs\, ihs) \end{aligned}$$

The substitutivity theorem for generic equality says that if two element are tested equal then they are indistinguishable:

$$\mathrm{subst\_eq}_{\Sigma} : (a : \mathrm{T}_{\Sigma}) \to (b : \mathrm{T}_{\Sigma}) \to |\mathrm{eq}_{\Sigma}\, a\, b| \to \mathrm{EQ}_{\mathrm{T}_{\Sigma}}\, a\, b$$

where $\mathrm{EQ}_X\, x\, y = (P : X \to \mathrm{Set}) \to P\, x \to P\, y$. The proof of this theorem follows exactly the same pattern as the proof of reflexivity and can be found on this paper's home page. Combining the generic definitions of equality, reflexivity and substitutivity we obtain a generic datoid-definition.

2.2 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:

$$\begin{aligned} \mathrm{NilNat} &: \mathrm{ListNat} \\ \mathrm{ConsNat} &: \mathrm{Nat} \to \mathrm{ListNat} \to \mathrm{ListNat} \end{aligned}$$

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

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

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

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

(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 Nil 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

$$\Sigma_{\mathrm{ListNat}} = [\mathrm{Nil}, \mathrm{NonRec}\ \Sigma_{\mathrm{Nat}}\ (\mathrm{Rec}\ \mathrm{Nil})]$$

where $\Sigma_{\mathrm{Nat}} = [\mathrm{Nil}, \mathrm{Rec}\ \mathrm{Nil}]$.

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:

$$\begin{aligned} \mathrm{F}_{[\alpha_1,\ldots,\alpha_n]}\ X &= \mathrm{F}^{\mathrm{ar}}_{\alpha_1}\ X + \ldots + \mathrm{F}^{\mathrm{ar}}_{\alpha_n}\ X \\ \mathrm{F}^{\mathrm{ar}}_{\mathrm{Nil}}\ X &= \mathbf{1} \\ \mathrm{F}^{\mathrm{ar}}_{\mathrm{Rec}\ \alpha}\ X &= X \times \mathrm{F}_\alpha\ X \\ \mathrm{F}^{\mathrm{ar}}_{\mathrm{NonRec}\ \Sigma\ \alpha}\ X &= \mathrm{T}_\Sigma \times \mathrm{F}_\alpha\ X \end{aligned}$$

As an example we instantiate the definition of F to obtain the expected pattern functor for ListNat:

$$\mathrm{F}_{\underline{\mathrm{ListNat}}}\ X = \mathbf{1} + (\mathrm{T}_{\underline{\mathrm{Nat}}} \times (X \times \mathbf{1}) + \mathbf{0}) \simeq \mathbf{1} + \mathrm{Nat} \times X$$

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 is available on the paper's home page.

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 (using extensional equality) derive the rules for one-sorted term algebras from the rules for iterated inductive definitions.

2.3 Infinitary 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 possibly infinitary operations, although working in classical set theory, his arities are possibly infinite ordinal numbers.)

We keep $\mathrm{Sig} = [\mathrm{Arity}]$ as in the homogeneous case, but we change Arity to be Set and modify the pattern functor:

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

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 higher number classes.

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

$$\begin{aligned} \mathrm{eq}_\Sigma &: \mathrm{T}_\Sigma \to \mathrm{T}_\Sigma \to \mathrm{Set} \\ \mathrm{eq}_\Sigma &= \mathrm{iter}_\Sigma\ \mathrm{eqstep}_\Sigma \\ \mathrm{eqstep}_\Sigma &: \mathrm{F}_\Sigma(\mathrm{T}_\Sigma \to \mathrm{Set}) \to \mathrm{T}_\Sigma \to \mathrm{Set} \\ \mathrm{eqstep}_\Sigma &\quad ps\ x = \mathrm{recog\_all}_\Sigma\, ps\, (\mathrm{out}_\Sigma\, x) \end{aligned}$$

where

$$\begin{aligned} &\mathrm{recog\_all}_\Sigma : \mathrm{F}_\Sigma\,(\mathrm{T}_\Sigma \to \mathrm{Set}) \to \mathrm{F}_\Sigma\,\mathrm{T}_\Sigma \to \mathrm{Set} \\ &\mathrm{recog\_all}_{I::\Sigma}\,(\mathrm{Inl}\, f)\,(\mathrm{Inl}\, x) = (i : I) \to f\, i\, (x\, i) \\ &\mathrm{recog\_all}_{I::\Sigma}\,(\mathrm{Inr}\, g)\,(\mathrm{Inr}\, y) = \mathrm{recog\_all}_\Sigma\, g\, y \\ &\mathrm{recog\_all}_{I::\Sigma}\,(\mathrm{Inl}\, f)\,(\mathrm{Inr}\, y) = \mathbf{0} \\ &\mathrm{recog\_all}_{I::\Sigma}\,(\mathrm{Inr}\, g)\,(\mathrm{Inl}\, x) = \mathbf{0} \end{aligned}$$

An interesting variation of this universe for infinitary inductive definition is obtained if we restrict branching to range over sets of the form T_Σ. Then we get the following notion of signature:

$$\begin{aligned}\mathrm{Sig} &= [\mathrm{Arity}]\\ \mathrm{Arity} &= \mathrm{Sig}\end{aligned}$$

If we present this definition with constructors (using the isomorphism between finitely branching trees and binary trees) we get

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

The pattern functor is:

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

In this setting we have

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

Note that neither of these two variants of universes for infinitary induction can capture iterated induction in the sense of 2.2. The branching (the number of arguments to one constructor) can be infinite here but the arity (the number of constructors) is finite. But we could also add a constructor NonRec for side conditions and thus combine infinitary and iterated induction into one universe.

3 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. The decoding function thus takes a signature and returns a parameterized term algebra, that is, it is a function

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

For simplicity, we present a universe for one-sorted term algebras with parameters – essentially the same as the one introduced by Pfeifer & Rueß [PR99]. If we add iterated induction we obtain the case considered in Jansson & Jeuring [JJ97]. It is of course also possible to consider parameterized infinitary induction.

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

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

and the set $\mathrm{Maybe}\,A$ with constructors

$$\begin{aligned}\mathrm{Nothing} &: (A : \mathrm{Set}) \to \mathrm{Maybe}\,A \\ \mathrm{Just} &: (A : \mathrm{Set}) \to A \to \mathrm{Maybe}\,A\end{aligned}$$

The universe construction. Compared with the homogeneous case we add a new constructor, Par, for arities[1]

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

The signature for parametric lists $[A]$ and for $\mathrm{Maybe}\,A$ are then

$$\begin{aligned}\Sigma_{[\cdot]} &= [\mathrm{Nil}, \mathrm{Par}\,(\mathrm{Rec}\,\mathrm{Nil})] \\ \Sigma_{\mathrm{Maybe}} &= [\mathrm{Nil}, \mathrm{Par}\,\mathrm{Nil}]\end{aligned}$$

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

$$\begin{array}{ccc} \mathrm{F}_\Sigma\,A\,(\mathrm{T}_\Sigma\,A) & \xrightarrow{\mathrm{Intro}_\Sigma\,A} & \mathrm{T}_\Sigma\,A \\ \Big\downarrow{\scriptstyle \mathrm{F}_\Sigma\,A\,(\mathrm{iter}_\Sigma\,A\,C\,d)} & & \Big\downarrow{\scriptstyle \mathrm{iter}_\Sigma\,A\,C\,d} \\ \mathrm{F}_\Sigma\,A\,C & \xrightarrow[d]{} & C \end{array}$$

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

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

The diagram for induction is modified accordingly.

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

$$\mathrm{Par} : \mathrm{Fin}\,n \to \mathrm{Arity}_n \to \mathrm{Arity}_n$$

Generic programs for parameterized types. 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 $A, B, C : \mathrm{Set}$.)

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

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

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

The application $\mathrm{zipWith}_\Sigma\ op\ x\ y$ succeeds iff x and y have the same structure and op succeeds for all pairs of corresponding elements. The result has the same structure as x and y and contains the results from the successful applications of the operator op. The general type of function $\mathrm{zipWith}_\Sigma$ is best explained through its instances. With $C = A \times B$ we obtain the familiar zip function from generic functional programming:

$$\begin{aligned}
\mathrm{zip}_\Sigma\ &:\ (A, B : \mathrm{Set}) \to \mathrm{Zip}\ A\ B\ (A \times B) \\
\mathrm{zip}_\Sigma\ A\ B &= \mathrm{zipWith}_\Sigma\ A\ B\ (A \times B)\ (\lambda x\ y.\ \mathrm{Just}\ (x, y))
\end{aligned}$$

With $C = \mathbf{1}$ we note that $\mathrm{Maybe}\ \mathbf{1} = \mathrm{Bool}$ and thus, the parameterized equality test eqBy_Σ can also be seen as a special case of $\mathrm{zipWith}_\Sigma$:

$$\begin{aligned}
\mathrm{Eq}\ A = A \to A \to \mathrm{Bool} = A \to A \to \mathrm{Maybe}\ \mathbf{1} &= \mathrm{Zip}\ A\ A\ \mathbf{1} \\
\mathrm{eqBy}_\Sigma\ A &\simeq \mathrm{zipWith}_\Sigma\ A\ A\ \mathbf{1}
\end{aligned}$$

4 Indexed inductive definitions

4.1 Many-sorted term algebras

First we shall consider the special case of many-sorted term algebras, giving rise to a simple class of mutually inductive definitions. See also Capretta [Cap99] for some other approaches to defining many-sorted term algebras in dependent type theory. This is the main class of term algebras considered in algebraic specification theory, following the work by the ADJ-group [GTW78].

For simplicity we first consider many-sorted algebras with finitely many sorts, and no parameters (it is easy to add them). Note that the iterated inductive definitions in section 2.2 are subsumed by the mutual inductive definitions here.

The type of signatures for n-sorted algebras is now

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

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

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

Another example is the mutual inductive definition of trees and forests. More generally, abstract syntax trees for context-free grammars are many-sorted algebras.

The diagram for initial n-sorted algebras is

$$\begin{array}{ccc} \mathrm{F}_{n,\Sigma}\, \mathrm{T}_{n,\Sigma}\, i & \xrightarrow{\mathrm{Intro}_{n,\Sigma}\, i} & \mathrm{T}_{n,\Sigma}\, i \\ \Big\downarrow{\scriptstyle \mathrm{F}_{n,\Sigma}\, (\mathrm{iter}_{n,\Sigma}\, d)\, i} & & \Big\downarrow{\scriptstyle \mathrm{iter}_{n,\Sigma}\, d\, i} \\ \mathrm{F}_{n,\Sigma}\, C\, i & \xrightarrow[d\, i]{} & C\, i \end{array}$$

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

We neither display the diagram for the full elimination (induction) rule which is similar to the one for the non-indexed case 2.1, nor give the definition of generic size and equality. Instead we move on the the more general case of inductive families.

4.2 Finitary indexed induction

In this section we consider a bigger class of finitary indexed inductive definitions. For simplicity, we choose to present the class of *restricted* indexed inductive definitions, rather than the class of *general* indexed inductive definitions, in the sense of Dybjer & Setzer [DS01]. To explain the difference we consider the Nat-indexed inductive definition of vectors (with elements of some fixed set A

for simplicity). This is most naturally presented as a general indexed inductive definition:

$$\begin{aligned} \mathrm{NilV} &: \mathrm{Vect}\,0 \\ \mathrm{ConsV} &: (n : \mathrm{Nat}) \to (x : A) \to \mathrm{Vect}\,n \to \mathrm{Vect}\,(\mathrm{Succ}\,n) \end{aligned}$$

We can reformulate this as a restricted indexed inductive definition, by employing an equality test for natural numbers:

$$\begin{aligned} \mathrm{NilV} &: (m : \mathrm{Nat}) \to (m = 0) \to \mathrm{Vect}\,m \\ \mathrm{ConsV} &: (m : \mathrm{Nat}) \to (n : \mathrm{Nat}) \to (m = \mathrm{Succ}\,n) \to (x : A) \to \mathrm{Vect}\,n \to \mathrm{Vect}\,m \end{aligned}$$

Restricted indexed inductive definitions require that the index in the result type is a variable.

Restricted indexed inductive definitions have some theoretical and practical advantages, but the drawback is that they give rise to longer and less natural formulation of the rules. The reader is referred to Dybjer & Setzer [DS01] for more discussion.

The universe construction. We define the universe $I \to \mathrm{Sig}_I$ for restricted I-indexed inductive definitions:

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

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.

Note that arities and signatures have been fused into one code type: Sig_I. The added power in the NonRec case can be used to build up what corresponds to the list of arities in simpler universes. A choice between n constructors can be coded by $\mathrm{NonRec}\,(\mathrm{Fin}\,n)\ \mathit{constrs}$ where $\mathit{constrs} : \mathrm{Fin}\,n \to \mathrm{Sig}_I$ gives the arity for each constructor.

An inductive family is a simultaneous definition of an indexed family of datatypes. In the special case when the set is finite the family can be coded as a group of mutually recursive datatypes, that is, as a many-sorted term algebra (Section 4.1). We get n-sorted algebras if $I = \mathrm{Fin}\,n$, if arities are only built up by Nil and Rec, and where NonRec is used at the top level for building up lists of arities.

To define the object part of the pattern functor

$$\mathrm{F}_{I,\Sigma} : (I \to \mathrm{Set}) \to (I \to \mathrm{Set})$$

for $\Sigma : I \to \mathrm{Sig}_I$ on the category of I-indexed families of sets, we introduce an auxiliary operator

$$\mathrm{G}_{I,\gamma} : (I \to \mathrm{Set}) \to \mathrm{Set}$$

for $\gamma : \mathrm{Sig}_I$. Then

$$\mathrm{F}_{I,\Sigma}\, X\, i = G_{I,\Sigma\, i}\, X$$

and G_γ is defined by induction on $\gamma : \mathrm{Sig}_I$:

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

The initial algebra diagram looks the same as in the many-sorted case. The type-theoretic rules are (with $\Sigma : I \to \mathrm{Sig}_I$):

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

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

A code for binary search trees. An example of an inductive family 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

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

Written as "arities" they become

$$\begin{aligned}
&\mathrm{arityBST}\, (lb, ub)\, 0 = \mathrm{NonRec}\, (lb < ub)\, (\lambda p.\mathrm{Nil}) \\
&\mathrm{arityBST}\, (lb, ub)\, 1 = \mathrm{NonRec}\, \mathrm{Nat}\, (\lambda root. \\
&\qquad\qquad \mathrm{NonRec}\, (lb < root)\, (\lambda p_1. \\
&\qquad\qquad\quad \mathrm{NonRec}\, (root < ub)\, (\lambda p_2. \\
&\qquad\qquad\qquad \mathrm{Rec}\, (lb, root)\, (\mathrm{Rec}\, (root, ub)\, \mathrm{Nil}))))
\end{aligned}$$

Thus the signature for the family BST becomes the family of codes Σ_{BST}:

$$\begin{aligned}
\Sigma_{\mathrm{BST}} &: \mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Sig}_{\mathrm{Nat} \times \mathrm{Nat}} \\
\Sigma_{\mathrm{BST}} &= \lambda bounds.\, \mathrm{NonRec}\, \mathbf{2}\, (\mathrm{arityBST}\, bounds)
\end{aligned}$$

Indexed generic functions. We can now write a generic size function (or rather, an indexed family of size functions) 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 been able to extend the definition of $\mathrm{zipWith}_{I,\Sigma}$ from section 3. We refer to the Alfa-implementation on the home page for details.

4.3 Infinitary indexed inductive definitions

In each of sections 2.1–4.2 we have presented a universe consisting of a set (family) of signatures and for each signature a term algebra. Each section defines a theory (a version of Martin-Löf type theory with a particular collection of inductive definitions) by adding some constants (with their types) and equations to the logical framework from section 1. The theory for one-sorted term algebras is given in Figure 2, and each of the other theories can be obtained by changing the axioms as described in the respective (sub)sections. 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 a large theory which can swallow all the previous theories. For this purpose we could use the the theory of indexed inductive-recursive definitions $\mathbf{IIR}^{\mathbf{ext}}$ (with extensional equality) given by Dybjer and Setzer [DS01]. In this theory we conjecture that all of our universes can be defined. To actually work out these embeddings in detail is however a task outside the scope of this paper.

In fact, since induction-recursion does not play a role in this paper, it suffices with the theory of indexed inductive definitions $\mathbf{IID}^{\mathbf{ext}}$ (with extensionality). **IID** is a natural upper bound of the theories presented in sections 2.1–4.2.

IID is just like the theory of finitary indexed inductive definitions in the previous subsection, except that we now have infinitary inductive definitions. Formally, this means that we generalize the case of a recursive premise. It becomes

$$\mathrm{Rec} : (A : \mathrm{Set}) \to (A \to I) \to \mathrm{Sig}_I \to \mathrm{Sig}_I$$

where the definition of G for the recursive case becomes

$$\mathrm{G}_{\mathrm{Rec}\,A\,i\,\gamma}\,X = ((x : A) \to X\,(i\,x)) \times \mathrm{G}_\gamma\,X$$

As an example of an infinitary indexed inductive definition we consider the accessible (or well-founded) part of a relation $<$ on a set I. The formation and introduction rules are

$$\begin{aligned}\mathrm{Acc} &: I \to \mathrm{Set}\\ \mathrm{AccIntro} &: (i : I) \to ((j : I) \to (j < i) \to \mathrm{Acc}\,j) \to \mathrm{Acc}\,i\end{aligned}$$

The signature for Acc is

$$\begin{aligned}\Sigma_{\mathrm{Acc}} &: I \to \mathrm{Sig}_I\\ \Sigma_{\mathrm{Acc}} &= \lambda i.\mathrm{NonRec}\,((j : I) \times (j < i))\,\mathrm{fst}\,\mathrm{Nil}\end{aligned}$$

We refer to [DS01,DS03a] for a full explanation of the theory **IIR** (and thus implicitly of its subtheory **IID**).

IID is a suitable general framework for generic programming, since we conjecture that the theories in Sections 2.1–4.2 are definable in **IID** in the following senses. (We have however not yet given a a rigorous proof of this conjecture.) Firstly, the set of signatures for one-sorted algebras (possibly with iterated induction) has a code in $\mathrm{Sig}_\mathbf{1}$ in $\mathbf{IID}^\mathbf{ext}$. Moreover, each signature for one-sorted algebras can be mapped to a signature in $\mathrm{Sig}_\mathbf{1}$, and the decoding function can be obtained by composing the decoding function for $\mathrm{Sig}_\mathbf{1}$ with this map. Furthermore the set of signatures for parameterized term algebras also has a code in $\mathrm{Sig}_\mathbf{1}$. Here a code in can be mapped to a function $\mathrm{Set} \to \mathrm{Sig}_\mathbf{1}$, and the decoding can again be obtained by composing the decoding function for $\mathrm{Sig}_\mathbf{1}$ with this map. We conjecture that similar embeddings can be done also for the theory of many-sorted term algebras and for the theory of finite indexed inductive definitions. The situation with infinitary induction in section 4.2 is similar to the situation with one-sorted algebras, except that as it stands the type of signatures is here a "large" inductive definition, since it has a constructor which refers to Set. This size problem can be solved if we replace the current large inductive definitions with an analogous small one.

5 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 3).

In Generic Haskell [HJ03] (the successor of PolyP) the universe is generalized to include mutually recursive and nested datatypes, as well as datatypes with parameters of higher kinds. This allows the full class of Haskell datatypes to be expressed but also restricts the set of definable generic functions. (The function

subterms : $\mathrm{T}_\Sigma \to [\mathrm{T}_\Sigma]$ is an example of a function which Generic Haskell cannot define simply because the concept of subterm is not meaningful for this general class of datatypes.)

Many datatypes with invariants can be simulated in Haskell using nested and datatypes with parameters of higher kinds, but these types can be more directly expressed using dependent types.

Combining dependent types and generic programming. The research on this topic comes from 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. 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 of parameterized term algebras. Others include universes for infinitary 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. We have worked out some more examples: proofs of reflexivity and substitutivity of generic equality. As the reader has seen, these proofs are non-trivial! To facilitate generic proofs we provide an elimination constant which captures primitive recursion rather than iteration.

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,DS03b,DS03a] 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 their papers have meta-theoretic rather than practical interest. Furthermore, for the purpose of practical generic programming the universe of inductive-recursive definitions is too large. (Not even Boolean equality can be defined over that universe.) 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.

[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.

[DS01] 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.

[DS03a] P. Dybjer and A. Setzer. Indexed induction-recursion. long version, submitted for publication, available from `http://www.cs.chalmers.se/~peterd/`, 2003.

[DS03b] P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 2003. In press.

[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.

[HJ03] 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, 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.