Valid XML Transformations with Dependent Types

Marcin Benke*

October 15, 2007

Abstract

Most XML processing languages and libraries ensure only that the generated XML document will be well-formed. In this paper we propose a method of defining XML transformations that ensure validity preservation.

To this end, we use a dependently typed framework as well as a method of defining universes, originally described in [BDJ03] for generic programs and proofs.

Introduction

Correctness of XML documents is defined in two steps: well-formed XML documents are basically those that conform to XML syntax. Validity on the other hand is a more complex issue. To quote from XML definition:

> "The function of the markup in an XML document is to describe its storage and logical structure and to associate attribute name-value pairs with its logical structures. XML provides a mechanism, the document type declaration, to define constraints on the logical structure and to support the use of predefined storage units. [...] An XML document is valid if it has an associated document type declaration and if the document complies with the constraints expressed in it."

Most XML processing languages and libraries ensure only that the generated XML document will be well-formed. In this paper we propose a method of defining XML transformations that ensure validity preservation.

To this end, we use a dependently typed framework as well as a method of defining universes, originally described in [BDJ03] for generic programs and proofs.

*Partially supported by Polish KBN grant 3 T11C 002 27

Plan of the paper First we describe briefly the framework we work in. Then we recapitulate the method of universe construction which constitutes the foundation for our XML transformation system. Since XML document types can be related to multi-sorted algebras, we describe the construction of universes for such algebras, while introducing the concepts on a simpler example of single-sorted algebras. Finally we define an universe for XML documents

The universes have been implemented and tested in Agda, which is basically a proof assistant, but also a dependently-typed functional programming language. To be able to test our universes, and later use them in practice, we have implemented Alonzo — a compiler for Agda. The final part of the paper describes briefly the metod of translation used in this compiler.

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 $\mathrm{El}\, A$ of elements for each $A : \mathrm{Set}$.

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 $\mathrm{El}\, A$ (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 $\mathrm{Inl}, \mathrm{Inr}$. 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 $\mathrm{Fin}\, n$ for the finite type with n elements denoted by $0, \ldots, n-1$. Formally, $\mathrm{Fin}\, 0 = \mathbf{1}$ and $\mathrm{Fin}\,(m+1) = \mathbf{1} + \mathrm{Fin}\, m$. 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:

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

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:

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

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 Sig = [Arity] = [Nat], and the decoding function T : Sig → 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.

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

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

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

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 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

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

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

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

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:

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

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. 1 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 $\mathrm{T}_{\underline{\mathrm{Nat}}} = \mathrm{Nat}$.

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

$$
\begin{array}{lcl}
\mathrm{Arity} & : & \mathrm{Type} \\
\mathrm{Zero} & : & \mathrm{Arity} \\
\mathrm{Succ} & : & \mathrm{Arity} \to \mathrm{Arity}
\end{array}
\qquad
\begin{array}{lcl}
\mathrm{Sig} & : & \mathrm{Type} \\
[\,] & : & \mathrm{Sig} \\
(::) & : & \mathrm{Arity} \to \mathrm{Sig} \to \mathrm{Sig}
\end{array}
$$

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

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

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

$$
\begin{array}{lcl}
\mathrm{T} & : & \mathrm{Sig} \to \mathrm{Set} \\
\mathrm{Intro} & : & (\Sigma : \mathrm{Sig}) \to \mathrm{F}^{0}_{\Sigma}\, \mathrm{T}_{\Sigma} \to \mathrm{T}_{\Sigma} \\
\mathrm{rec} & : & (\Sigma : \mathrm{Sig}) \to (C : \mathrm{T}_{\Sigma} \to \mathrm{Set}) \to \\
 & & ((y : \mathrm{F}^{0}_{\Sigma}\, \mathrm{T}_{\Sigma}) \to \mathrm{F}^{\mathrm{IH}}_{\Sigma}\, \mathrm{T}_{\Sigma}\, C\, y \to C\, (\mathrm{Intro}_{\Sigma}\, y)) \to \\
 & & (x : \mathrm{T}_{\Sigma}) \to C\, x \\
\multicolumn{3}{l}{\mathrm{rec}_{\Sigma}\, C\, d\, (\mathrm{Intro}_{\Sigma}\, y) = d\, y\, (\mathrm{F}^{\mathrm{map}}_{\Sigma}\, \mathrm{T}_{\Sigma}\, C\, (\mathrm{rec}_{\Sigma}\, C\, d)\, y)}
\end{array}
$$

Figure 1: Axioms for the theory of homogeneous term algebras (large elimination rules can be added)

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

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

Figure 2: Elimination rules for arities and signatures. (Again, large elimination rules can be added.)

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:

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

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. 2), but in this presentation we use pattern matching and explicit recursion for readability.

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

$$
\mathrm{size}_\Sigma \quad = \quad \mathrm{iter}_\Sigma\ \mathrm{sizestep}_\Sigma
$$

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

where

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

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

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

$$\begin{array}{lll}\text{sizestep}_{\underline{\text{Nat}}} : \mathbf{1} + (\text{Nat} \times \mathbf{1} + \mathbf{0}) & \to & \text{Nat}\\ \text{sizestep}_{\underline{\text{Nat}}}\ (\text{Inl}\,()) & = & 1\\ \text{sizestep}_{\underline{\text{Nat}}}\ (\text{Inr}\,(\text{Inl}\,(\mathit{subsum}, ()))) & = & 1 + \mathit{subsum}\end{array}$$

Note that this means that $\text{size}\, n = n + 1$ 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:

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

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

$$\text{eqstep}_\Sigma\, x\, (\text{Intro}\, y) = \text{recog\_all}_\Sigma\, \text{T}_\Sigma\, x\, y$$

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

Formally;

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

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

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

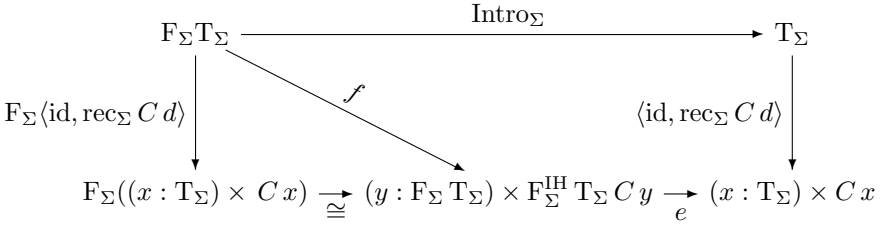
$$
\begin{array}{llllll}
\mathrm{recog\_all}_\Sigma : (X : \mathrm{Set}) \to \mathrm{F}_\Sigma\,(X \to \mathrm{Bool}) \to \mathrm{F}_\Sigma\,X \to \mathrm{Bool} \\
\mathrm{recog\_all}_{n::\Sigma} & X & (\mathrm{Inl}\,fs) & (\mathrm{Inl}\,xs) & = & \mathrm{and\_args}_n\,X\,fs\,xs \\
\mathrm{recog\_all}_{n::\Sigma}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

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

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

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

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

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

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

as in Dybjer & Setzer. Their formal definitions can be found in Fig. 1.

Hence the elimination rule is

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

The equality rule is

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

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.

3 Indexed inductive definitions

3.1 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).

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:

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

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.

3.2 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:

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

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 3.1). 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

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

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

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

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

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

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

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

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

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

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

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

Written as "arities" they become

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

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

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

Indexed generic functions. We can now write a generic size function (or rather, an indexed family of size functions) over this universe

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

However, to define equality

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

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

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

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

3.3 Infinitary indexed inductive definitions

In each of sections 2.1–3.2 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 1, 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 2.1–3.2.

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

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

where the definition of G for the recursive case becomes

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

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

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

The signature for Acc is

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

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 2.1–3.2 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 Set $\rightarrow \mathrm{Sig}_{\mathbf{1}}$, 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 3.2 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.

4 Universes for XML Documents

4.1 Micro-XML

To see how an universe for XML documents can be constructed, let us first consider a variant of XML (which we'll call Micro-XML) which is XML subject to following restrictions:

- There are no tag attributes
- For every element there is a unique sequence of elements that is its valid content (hence there is no CTYPE content and no entities)

Micro-XML retains enough of XML structure to be interesting, yet is simple enough for us to present the idea clearly and without unnecessary clutter.

A universe for Micro-XML can be constructed in a manner similar to the one for many-sorted algebras, presented in Section 3.1:

```
Tag : Set
Tag = String

Code : Set
Code = List Tag

DTD : Set
DTD = Tag -> Code

F : (d : DTD) -> (X : Tag -> Set) -> Tag -> Set
F d X t = Fc (X t) where
  Fc [] = Unit
  Fc (t:ts) = Pair (X t) (Fc ts)

data T(d : DTD)(t :Tag) : Set where
  In F d (T d) t -> T d t
```

```
It (d : DTD) (C : Tag -> Set) (d : (t : Tag) -> F d C t -> C t)
   -> T d t -> C t

data Elem(dtd : DTD) : Tag -> Set where
  Node : (t : Tag) -> T dtd (dtd t) -> Elem dtd t
```

4.2 Universe for XML documents

Now that the general idea has been laid out in the previous section, we can construct a universe for full XML. In the contruction below, we omit element attributes, which does not reduce generality, but makes for a cleaner presentation.

```
Tag : Set
Tag = String

data ElemCode : Set where
  CString : ElemCode
  CEl : Tag -> ElemCode
  CSeq : ElemCode -> ElemCode -> ElemCode
  CPlus : ElemCode -> ElemCode
  CAlt : ElemCode -> ElemCode -> ElemCode
  CBad : ElemCode

DTD : Set
DTD = Tag -> ElemCode

LDTD : Set
LDTD = (List Char) -> ElemCode

liftDTD : LDTD -> DTD
liftDTD ld tag = ld (toList tag)

F : DTD -> ElemCode -> (X : ElemCode -> Set) -> Set
F dtd CString      X = String
F dtd (CEl tag)    X = X (dtd tag)
F dtd (CAlt c1 c2) X = Either (X c1)(X c2)
F dtd CBad         X = False

F1 : (dtd : DTD) -> (c : ElemCode)
     -> (X : ElemCode -> Set)
     -> (Y : ElemCode -> Set)
     -> (f : (d : ElemCode) -> X d -> Y d)
     -> F dtd c X
     -> F dtd c Y
```

```
F1 dtd CString X Y f s = s
F1 dtd (CEl t) X Y f x = f (dtd t) x
F1 dtd (CAlt c1 c2) X Y f (Left l) = Left (f c1 l)
F1 dtd (CAlt c1 c2) X Y f (Right r) = Right (f c2 r)

data T(d : DTD)(c : ElemCode) : Set where
 In : F d c (T d) -> T d c

It : (dtd : DTD) -> (c : ElemCode)
   -> (d : (c' : ElemCode)  ...

data Elem(dtd : DTD) : Tag -> Set where
  Node : (t : Tag) -> T dtd (dtd t) -> Elem dtd t
```

5 A Compiler for Agda

The universes described above have been implemented and tested in Agda, which is basically a proof assistant, but also a dependently-typed functional programming language. To be able to test our universes, and later use them in practice, we have implemented Alonzo — a compiler for Agda, translating it into Haskell, which is then compiled using GHC. The remainder of the paper describes the highlights of Alonzo design and implementation.

5.1 Translating Dependent Types into Haskell

At first glance, translating dependent types to Haskell type system is easy: just forget all the dependencies. For example, consider the inductive family of vectors and a function representing their concatenation:

```
data Vec (A : Set) : Nat -> Set where
  nil   : Vec A zero
  cons : {n : Nat} -> A -> Vec A n -> Vec A (suc n)

append : {A:Set} -> {n,m : Nat} -> Vec A n -> Vec A m -> Vec A (n+m)
...
```

One could translate them into following Haskell definitions:[1]

```
data List a where
  nil :: List a
  cons  :: List a -> List a -> List a

append : List a -> List a -> List a
```

[1] Although we use GHC's extended syntax here, the definitions below can easily be expressed in Haskell98 as well.

A similar approach was used by Brady in his Epigram compiler [Bra05].

However, things get more complicated when you get to large elimination. To illustrate the problem, consider the following definitions:

```
Q : Bool -> Set
Q true = Nat
Q false = Bool

f : (b:Bool) -> Q b
f true = pred 3
f false = true

mainS : String
mainS = showBool (f (const false true))
```

Note that the actual result type of f depends on the value of its arguments. How can such a function be translated into Haskell?

Assume we had a "magic" function `cast :: a -> b`. We could then translate `f` as follows:

```
f :: Bool -> b
f (true) = cast (pred (cast (3)))
f (false) = cast true
```

Luckily, GHC has such a function, albeit with an uglier name: `unsafeCoerce#`. Despite the name however, all coercions (typecasts, actually) we insert are safe, since the program has been already checked by Agda typechecker.

5.2 Translating Types

We need two translations for types:

1. to Haskell types (only data types)
2. to Haskell values

As for the first one, although it could seem that with the casts described above explicit use of Haskell types is not needed, we still need data types for pattern matching and of course as containers for data. But we don't really need to translate all the types, but only the datatypes, introduced with the `data` definitions.

This translation is relatively straightforward. Every Agda datatype is translated to a Haskell datatype and every constructor is translated to a constructor of the same arity.

However, in this translation we sacrifice a bit of (potential) efficiency for safety: GHC could potentially use the (not necessarily correct) knowledge of the types for optimisations. To prevent this,

```
data List (A:Set) : Set where
   nil : List a
   cons : A -> List A -> List A
```

becomes

```
data List a b = C1 | C2 a b
```

5.3 Translating Types to Values

In a dependently-typed system, types can be used as values in expressions. In Haskell no such thing is possible, so we must find a way of translating Agda types to Haskell values. One could of course argue that types cannot influence the result of the computation on the value level, and thus could be erased. However such erasure would alter arity of functions and quite often also strictness properties of the program. This seems to be a sufficient reason to avoid such erasure and go the extra mile of translating types to values as well.

How to translate

```
Q : Bool -> Set
Q true = Nat
Q false = Bool
```

into Haskell?

We use codes (actually, universes, if one prefers a fancier term); all datatypes (as well as primitive types) will have a value of this type assigned at compilation time, e.g.

```
dQ :: PreludeBool.Bool -> Runtime.Code
dQ (PreludeBool.C1) = cast PreludeNat.dNat
dQ (PreludeBool.C2) = cast PreludeBool.dBool
```

5.4 Codes

Since there is no pattern-matching on types (only on values), the encoding doesn't need to be injective. Hence, if we don't care for types at runtime, and want every bit of efficiency, a very simple coding can be used:

```
dNat = ()
dBool = ()
...
```

In this encoding the type of codes is simply the unit type; every type is encoded as unit value. This has the advantage of avoiding type computation at runtime entirely, while still preserving arities and strictness properties.

On the other hand, if we want run-time type information, we can introduce a more refined datatype of codes (e.g. the universe for dependent types described in [BDJ03]. Then we could have e.g. a (built-in) function

```
showSet : Set -> String
```

at very little cost.

5.5 Pattern matching

Another obstacle is pattern matching, consider an Agda definition

```
printf' : (fmt : List Format) -> Printf' fmt -> String
printf' (stringArg   :: fmt) < s | args > = s ++ printf' fmt args
...
printf' (badFormat _ :: fmt) ()
printf'  []  unit      = ""
```

The patterns are of different types, whereas Haskell demands that in all clauses of a definition, patterns for a given argument be of the same type. Casts don't solve the problem we cannot cast patterns.

We can solve this problem by making every clause into a separate single-clause definition and then gathering them together as local definitions for the main clause; for example the definition above would be translated as

```
d38 = d38_1
where d38_1 (PreludeList.C4 (Printf4.C3) v0)
        (AlonzoPrelude.C43 v1 v2)
        = cast
            (PreludeString.d0 (cast v1)
               (cast (Printf4.d38 (cast v0) (cast v2))))
      d38_1 a b = cast d38_2 a b
...
d38_8 (PreludeList.C3) (Printf4.C1) = cast ("")
```

References

[BDJ03] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

[Bra05] Edwin C. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

[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.

[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.

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