Towards generic programming in type theory

Marcin Benke

Department of Computing Science
Chalmers University of Technology
412 96 Göteborg, Sweden
e-mail: marcin@cs.chalmers.se

Abstract. We present a method of embedding generic programming within a dependent type theory in a manner supporting generic verification (i.e. generic proofs of generic properties of generic programs). Further we investigate how the idea of generic programming can be extended to inductive families.

1 Introduction

1.1 Context: proof editors and type theory

Alfa [HR00] is a graphical, syntax-directed editor for the proof system Agda [Coq98], is an implementation of *s*tructured type theory (STT) [CC99], which is based on Martin-Löf's type theory [ML84].

Like its predecessors in the ALF family of proof editors [Mag94], Alfa allows one to, interactively and incrementally, define theories (axioms and inference rules), formulate theorems and construct proofs of the theorems. All steps in the proof construction are immediately checked by the system and no erroneous proofs can be constructed.

Alfa is a term-based proof editor. This means that the proof is presented and recorded as a term, rather than as a tactic expression as in tactic-based proof editors such as Coq and Lego. Of course this does not preclude usage of tactics.

Alternatively, you can view Alfa as a syntax-directed editor for a small purely functional programming language with a type system that provides dependent types, thus allowing specification and verification of program properties within its type system. In fact, the language is very similar to the functional language Cayenne [Aug98] by Lennart Augustsson (which in turn is based on Haskell). As such, Alfa supports algebraic datatypes, pattern matching and general recursive definitions.

1.2 Generic Programming

Generic (also called polytypic) programming is an idea that has in recent time attracted a lot of attention, both in programming [Jan00] and proof editor communities [PR99]. Extensions facilitating generic programming has been proposed for languages such as Haskell (e.g. PolyP language of Jansson and Jeuring [JJ98] and Generic Haskell [CHJ+01,HJ]).

Type theory however, is a system powerful enough to express concepts of generic programming without any extensions. One way to do it is to explain inductive datatypes using an universe of shapes (cf e.g. [DS99], even though the universe presented here is different to one proposed by Dybjer and Setzer).

1.3 Related work

Pfeifer and Rueß [PR99] also use type theory to represent some polytypic programs, but their method doesn't seem general or extensible enough, nor does it seem suitable for generic verification.

Recently, Altenkirch and McBride [AM02] have presented a method of implementing generic programming in OLEG — a rearrangement of Pollack's LEGO system with some support for programming. Apart from different methods, our work puts more emphasis on creating **generic proofs** of properties of generic functions (generic verification), rather than the generic programming itself, while Altenkirch and McBride focus more on the faithful reflection of the higher-kinded nature of Generic Haskell. Also, with our approach, while codes of algebraic datatypes are used as an intermediate step, we can state and prove not only generic theorems about the codes, but also about the native datatypes of the framework.

2 Parameterless regular types

To explain the basic ideas of our approach let us start by discussing the representation of parameterless regular types. This case, while not particularly exciting seems to be a good opportunity to present the kernel of our method.

2.1 Shapes

The universe of shapes of algebraic types without parameters can be defined as follows

$Shape0 \in Set$

$$Shape0 \equiv \mathbf{data} \left\{ \begin{array}{l} \mathbf{Unit} \\ \mathbf{Rec} \\ * \, (r1, r2 \in Shape0) \\ + \, (r1, r2 \in Shape0) \end{array} \right\}$$

$Image0 \, (S \in Set, s \in Shape0) \in Set$

$$\begin{array}{l} Image0 \, S \, \mathbf{Unit} \equiv () \\ Image0 \, S \, \mathbf{Rec} \equiv S \\ Image0 \, S \, (r1 * r2) \equiv Image0 \, S \, r1 \wedge Image0 \, S \, r2 \\ Image0 \, S \, (r1 + r2) \equiv Image0 \, S \, r1 \vee Image0 \, S \, r2 \end{array}$$

2.2 Regular0 signature

$Regular0 \, (S \in Set) \in Type$

$$Regular0 \, S \equiv \mathbf{sig} \left\{ \begin{array}{l} shape \in Shape0 \\ out \in S \to Image0 \, S \, shape \\ inn \in Image0 \, S \, shape \to S \\ inoutL \in (a \in S, P \in Pred \, S) \to P \, (inn \, (out \, a)) \to P \, a \\ inoutR \in (a \in S, P \in Pred \, S) \to P \, a \to P \, (inn \, (out \, a)) \end{array} \right\}$$

This Alfa signature corresponds somehow to Haskell class definition:

```
class Regular0 s where
  shape :: Shape0
  ...
```

However we make use of dependent types, e.g. the type of *out* depends on the value of *shape*.

2.3 Example: Nat

The type of natural numbers

$Nat \in Set$

$$Nat \equiv \mathbf{data} \left\{ \begin{array}{l} \mathbf{0} \\ \mathbf{S} \, (n \in Nat) \end{array} \right\}$$

can be viewed as a parameterless regular type:

$Reg0Nat \in Regular0 \, Nat$

$$
\mathit{Reg0Nat} \equiv \textbf{struct} \left\{ \begin{array}{l}
\mathit{shape} \equiv \textbf{Unit} + \textbf{Rec} \\
\mathit{image} \equiv \mathit{Image0}\,\mathit{Nat}\,\mathit{shape} \\
\mathit{out} \equiv \lambda\, n \;\to\; \textbf{case}\; n\; \textbf{of} \left\{ \begin{array}{l} \mathbf{0} \to \textbf{Left}\,() \\ \mathbf{S}\, n' \to \textbf{Right}\, n' \end{array} \right\} \\
\mathit{inn} \equiv \lambda\, h \;\to\; \textbf{case}\; h\; \textbf{of} \left\{ \begin{array}{l} \textbf{Left}\, a \to \mathbf{0} \\ \textbf{Right}\, b \to \mathbf{S}\, b \end{array} \right\} \\
\mathit{inoutL} \equiv \lambda\, a\, P\, h \;\to\; \textbf{case}\; a\; \textbf{of} \left\{ \begin{array}{l} \mathbf{0} \to h \\ \mathbf{S}\, n \to h \end{array} \right\} \\
\mathit{inoutR} \equiv \lambda\, a\, P\, h \;\to\; \textbf{case}\; a\; \textbf{of} \left\{ \begin{array}{l} \mathbf{0} \to h \\ \mathbf{S}\, n \to h \end{array} \right\}
\end{array} \right\}
$$

This structure definition corresponds somehow to Haskell instance definition:

```
instance Regular0 Nat where
  shape = ...
```

It is also a constructive proof that *Nat* is regular, where *inn* and *out* define the encoding, while *inoutL* and *inoutR* provide the proof that the encoding is indeed an isomorphism.

It is worth noting that while such structure definitions can get quite long for more complicated types, they can easily be generated automatically (in a manner similar to Haskell deriving mechanism).

2.4 Generic functions: 3-stage definition

Our method allows defining generic functions that work for every member of the *Regular0* class. This process usually involves three steps:

1. Define a function working on encodings by induction on shapes.
2. Given regular R, specialize it to $R.\mathit{shape}$ (however R remains a parameter).
3. Use the isomorphism to transfer the definition from encoding of R to R itself (again R remains a parameter).

For example, let us define generic (propositional) equality:

$$
\begin{array}{l}
\mathit{Reg0Eq1}(S \in \mathit{Set}, \mathit{shape} \in \mathit{Shape0}, \\
\qquad\quad a, b \in \mathit{Image0}\, S\; \mathit{shape}) \in \mathit{Set}
\end{array}
$$

$$
\mathit{Reg0Eq2}\; (S \in \mathit{Set}, R \in \mathit{Regular0}\, S) \in \mathit{Rel}\, (\mathit{Image0}\, S\, R.\mathit{shape})
$$

$$
\mathit{Reg0Eq2}\, S\, R \equiv \mathit{Reg0Eq1}\, S\, R\, R.\mathit{shape}
$$

$$\mathit{Reg0Eq3}\ (S \in \mathit{Set}, R \in \mathit{Regular0}\, S) \in \mathit{Rel}\, S$$

$$\mathit{Reg0Eq3}\, S\, R \equiv \lambda\, a\, b\ \rightarrow\ \mathit{Reg0Eq2}\, S\, R\ (R.\mathit{out}\, a)\ (R.\mathit{out}\, b)$$

2.5 Sample generic proof: substitutivity

We can also state and prove generic theorems about our generic functions. This can also be done in three steps corresponding to the steps of function definition.

For example let us prove that the equality relation defined in the previous section is substitutive:

$$
\begin{array}{l}
\mathit{Reg0subst1}\, (\mathit{shape} \in \mathit{Shape0}, a, b \in \mathit{Image0}\, S\, \mathit{shape}, P \in \mathit{Pred}\, (\mathit{Image0}\, S\, \mathit{shape}), \\
\qquad\qquad e \in\ \mathit{Reg0Eq1}\, S\, R\, \mathit{shape}\, a\, b, \mathit{pa} \in P\, a) \in P\, b \\
\qquad\qquad (S \in \mathit{Set}, R \in \mathit{Regular0}\, S) \\
\quad \mathit{Reg0subst1}\, \mathbf{Unit}\ \mathbf{()}\ \mathbf{()}\, P\, e\, \mathit{pa}\ \equiv\ \mathit{pa} \\
\quad \mathit{Reg0subst1}\, \mathbf{Rec}\, a\, b\, P\, e\, \mathit{pa} \equiv \mathit{Reg0substAdapter3}\, S\, R\ (\mathit{Reg0subst1}\, R.\mathit{shape})\ a\, b\, P\, e\, \mathit{pa} \\
\quad \mathit{Reg0subst1}\ (\mathit{r1}\ *\ \mathit{r2})\ \langle a_1, a_2\rangle\ \langle b_1, b_2\rangle\ P\ \langle e_1, e_2\rangle\ \mathit{pa} \\
\equiv \mathbf{let} \left[\begin{array}{l}
\mathit{P1} \in \mathit{Pred}\ (\mathit{Image0}\, S\, \mathit{r1}) \\
\mathit{P1}\ \equiv\ \lambda\, h\ \rightarrow\ P\ \langle h, b_2\rangle \\
\mathit{P2} \in \mathit{Pred}\ (\mathit{Image0}\, S\, \mathit{r2}) \\
\mathit{P2}\ \equiv\ \lambda\, h\ \rightarrow\ P\ \langle a_1, h\rangle \\
\mathit{h1} \in \mathit{P1}\, a_1 \\
\mathit{h1}\ \equiv\ \mathit{Reg0subst1}\, \mathit{r2}\, a_2\, b_2\, \mathit{P2}\, e_2\, \mathit{pa}
\end{array} \right. \\
\quad \mathbf{in}\, \mathit{Reg0subst1}\, \mathit{r1}\, a_1\, b_1\, \mathit{P1}\, e_1\, \mathit{h1} \\
\quad \mathit{Reg0subst1}\ (\mathit{r1}\ +\ \mathit{r2})\ (\mathbf{Left}\, a_1)\ (\mathbf{Left}\, b_1)\ P\, e\, \mathit{pa} \equiv \mathit{Reg0subst1}\, \mathit{r1}\, a_1\, b_1\, \lambda\, h\ \rightarrow \\
P\ (\mathbf{Left}\, h)\ e\, \mathit{pa} \\
\quad \mathit{Reg0subst1}\ (\mathit{r1}\ +\ \mathit{r2})\ (\mathbf{Right}\, a_2)\ (\mathbf{Right}\, b_2)\ P\, e\, \mathit{pa} \equiv \mathit{Reg0subst1}\, \mathit{r2}\, a_2\, b_2\, \lambda\, h\ \rightarrow \\
P\ (\mathbf{Right}\, h)\ e\, \mathit{pa}
\end{array}
$$

Most of the work has been done now, steps two and three just do more or less "administrative" work of transferring the result from codes to native types:

$$
\begin{array}{l}
\mathit{Reg0subst2}\ (S \in \mathit{Set}, R \in \mathit{Regular0}\, S) \in \\
\mathit{isSubstitutive}\ (\mathit{Image0}\, S\, R.\mathit{shape})\ (\mathit{Reg0Eq2}\, S\, R)
\end{array}
$$

$$\mathit{Reg0subst2}\, S\, R \equiv \mathit{Reg0subst1}\, R.\mathit{shape}$$

$$\mathit{Reg0subst3}\ (S \in \mathit{Set}, R \in \mathit{Regular0}\, S) \in \mathit{isSubstitutive}\, S\ \ (\mathit{Reg0Eq3}\, S\, R)$$

$$\mathit{Reg0subst3}\, S\, R \equiv \mathit{Reg0substAdapter3}\, S\, R\ \ (\mathit{Reg0subst2}\, S\, R)$$

$$\begin{aligned}&\mathit{Reg0substAdapter3}\,(S \in \mathit{Set}, R \in \mathit{Regular0}\,S,\\
&\qquad \mathit{sub0} \in \mathit{isSubstitutive}\ (\mathit{Image0}\,S\,R.\mathit{shape})\ (\mathit{Reg0Eq2}\,S\,R))\\
&\qquad \in \mathit{isSubstitutive}\,S\ (\mathit{Reg0Eq3}\,S\,R)\end{aligned}$$

$$\begin{aligned}&\mathit{Reg0substAdapter3}\,S\,R\,\mathit{sub0} \equiv\ \lambda\,a\,b\,P\ ab\,pa\ \to\\
&R.\mathit{inoutL}\,b\,P\ \ (\mathit{sub0}\ (R.\mathit{out}\,a)\ (R.\mathit{out}\,b)\ \lambda\,h\ \to\ P\ \ (R.\mathit{inn}\,h)\ \ ab\ (R.\mathit{inoutR}\,a\,P\,pa))\end{aligned}$$

3 Regular types with arbitrary parameter number

3.1 Universes

The universe of shapes of algebraic types with n parameters can be defined as follows

$$\mathit{Shape}\ (n\ \in\ \mathit{Nat}) \in \mathit{Set}$$

$$\mathit{Shape}\,n \equiv \mathbf{data}\left\{\begin{array}{l}()\\ \mathbf{Par}\,(m\ \in\ \mathit{Fin}\,n)\\ \mathbf{Rec}\\ *\,(r1, r2\ \in\ \mathit{Shape}\,n)\\ +\,(r1, r2\ \in\ \mathit{Shape}\,n)\end{array}\right\}$$

The above definition uses dependent types, e.g. the type $\mathit{Fin}\,n$ representing a finite set with n elements:

$$\mathit{Fin}\ (n\ \in\ \mathit{Nat}) \in \mathit{Set}$$

$$\begin{aligned}&\mathit{Fin}\,\mathbf{0} \equiv \bot\\
&\mathit{Fin}\ (\mathbf{S}\,n') \equiv \mathbf{data}\left\{\begin{array}{l}\mathbf{FZ}\\ \mathbf{FS}\,(f\ \in\ \mathit{Fin}\,n')\end{array}\right\}\end{aligned}$$

Another dependent type used in our construction is $\mathit{SetVec}\ n$ — vector of sets of length n:

$$\mathit{SetVec}\ \in\ \mathit{Nat}\ \to\ \mathit{Type}$$

$$\mathit{SetVec} \equiv \mathbf{data}\left\{\begin{array}{l}[] \in \mathit{SetVec}\,\mathbf{0}\\ \mathbf{:}\ \ (m\ \in\ \mathit{Nat}, A\ \in\ \mathit{Set}, f\ \in\ \mathit{SetVec}\,m) \in \mathit{SetVec}\,\mathbf{S}(m)\end{array}\right\}$$

Using this type we can reflect the algebraic types by constructing image of a set under a given shape:

$$\mathit{Image}\ (n\ \in\ \mathit{Nat}, S\ \in\ \mathit{Set}, SV\ \in\ \mathit{SetVec}\,n, s\ \in\ \mathit{Shape}\,n)\ \in\ \mathit{Set}$$

$$
\begin{aligned}
&\mathit{Image}\, n\, S\, SV\, \mathbf{()} \equiv \mathit{()}\\
&\mathit{Image}\, n\, S\, SV\ (\mathbf{Par}\ m) \equiv SVget\, n\, SV\, m\\
&\mathit{Image}\, n\, S\, SV\, \mathbf{Rec} \equiv S\\
&\mathit{Image}\, n\, S\, SV\ (r1\ *\ r2) \equiv \mathit{Image}\, n\, S\, SV\, r1\ \wedge\ \mathit{Image}\, n\, S\, SV\, r2\\
&\mathit{Image}\, n\, S\, SV\ (r1\ +\ r2) \equiv \mathit{Image}\, n\, S\, SV\, r1\ \vee\ \mathit{Image}\, n\, S\, SV\, r2
\end{aligned}
$$

The fact that some type is regular is witnessed by a structure of the signature *Regular*:

$$\mathit{Regular}\ (n\ \in\ \mathit{Nat}, \mathit{Ps}\ \in\ \mathit{SetVec}\, n, S\ \in\ \mathit{Set})\ \in\ \mathit{Type}$$

$$
\mathit{Regular}\, n\, \mathit{Ps}\, S \equiv \mathbf{sig}\ \left\{
\begin{array}{l}
\mathit{shape} \in \mathit{Shape}\, n\\
\mathit{image} \in \mathit{Set}\quad \texttt{\{- = Image n S Ps shape -\}}\\
\mathit{out} \in S\ \rightarrow\ \mathit{Image}\, n\, S\, \mathit{Ps}\, \mathit{shape}\\
\mathit{inn} \in \mathit{Image}\, n\, S\, \mathit{Ps}\, \mathit{shape}\ \rightarrow\ S\\
\mathit{inoutL} \in (a\ \in\ S, P\ \in\ \mathit{Pred}\, S)\ \rightarrow\ P\ (\mathit{inn}\ (\mathit{out}\, a))\ \rightarrow\ P\, a\\
\mathit{inoutR} \in (a\ \in\ S, P\ \in\ \mathit{Pred}\, S)\ \rightarrow\ P\, a\ \rightarrow\ P\ (\mathit{inn}\ (\mathit{out}\, a))
\end{array}
\right\}
$$

Such witnesses can be generated automatically by a mechanism similar to the *Derive* mechanism described in the previous section.

3.2 Example: List

$$\mathit{List}(a \in \mathit{Set}) = \mathbf{data}\ [\,]\ |\ \mathbf{:}(x \in a, xs \in \mathit{List}\ a)$$

$$\mathit{RegList}\ (a\ \in\ \mathit{Set}) \in \mathit{Regular}\ (\mathbf{S\,0})\ (a\mathbf{:}[\,])\ (\mathit{List}\, a)$$

$$
\begin{array}{l}
\mathit{RegList}\, a \equiv \mathbf{struct}\\
\quad\left\{
\begin{array}{l}
\left[\mathit{shape} \equiv \mathbf{()}\ +\ (\mathbf{Par}\, \mathbf{FZ}\ *\ \mathbf{Rec})\right.\\
\left[\mathit{image} \equiv \mathit{Image}\ (\mathbf{S\,0})\ (\mathit{List}\, a)\ (a\mathbf{:}[\,])\ \mathit{shape}\right.\\
\left[\mathit{out} \equiv \lambda\, l\ \rightarrow\ \mathbf{case}\ l\ \mathbf{of} \left\{\begin{array}{l} [\,]\ \rightarrow\ \mathbf{Left}\ \mathbf{()}\\ (x : xs) \rightarrow \mathbf{Right}\ \langle x, xs\rangle \end{array}\right\}\right.\\
\left[\mathit{inn} \equiv \lambda\, h\ \rightarrow\ \mathbf{case}\ h\ \mathbf{of} \left\{\begin{array}{l} \mathbf{Left}\ a'\ \rightarrow\ [\,]\\ \mathbf{Right}\ \langle x, xs\rangle\ \rightarrow\ (x\ \mathbf{:}\ xs) \end{array}\right\}\right.\\
\left[\mathit{inoutL} \equiv \lambda\, l\, P\, h\ \rightarrow\ \mathbf{case}\ \ l\ \mathbf{of} \left\{\begin{array}{l} [\,]\ \rightarrow\ h\\ (x\ \mathbf{:}\ xs)\ \rightarrow\ h \end{array}\right\}\right.
\end{array}
\right\}
\end{array}
$$

3.3 Example: Triple

$$\mathit{Triple}\ (A1, A2, A3 \in \mathit{Set}) \in \mathit{Set}$$

$$\mathit{Triple}\ A1\ A2\ A3 \equiv \mathbf{data}\ \left\{ \mathbf{TrI}\,(l \in A1, m \in A2, r \in A3) \right\}$$

$$\mathit{RegTriple}\ (A1, A2, A3 \in \mathit{Set}) \in$$
$$\mathit{Regular}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\,\mathbf{0})))\ [A1, A2, A3]\ (\mathit{Triple}\ A1\ A2\ A3)$$

$$\mathit{RegTriple}\ A1\ A2\ A3 \equiv \mathbf{struct}$$
$$\left\{ \begin{array}{l} \left[\ \mathit{shape} \equiv \mathbf{Par}\,\mathbf{FZ} * \mathbf{Par}\ (\mathbf{FS}\,\mathbf{FZ}) * \mathbf{Par}\ (\mathbf{FS}\ (\mathbf{FS}\,\mathbf{FZ})) \right. \\ \left[\ \mathit{image} \equiv \mathit{Image}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{S}\,\mathbf{0})))\ (\mathit{Triple}\ A1\ A2\ A3)\ [A1, A2, A3]\ \mathit{shape} \right. \\ \left[\ \mathit{out} \equiv \lambda\, t \to \mathbf{case}\ t\ \mathbf{of} \left\{ \mathbf{TrI}\,l\,m\,r \to \langle l, \langle m, r\rangle\rangle \right\} \right. \\ \left[\ \mathit{inn} \equiv \lambda\, h \to \mathbf{case}\ h\ \mathbf{of} \left\{ \langle l, \langle m, r\rangle\rangle \to \mathbf{TrI}\,l\,m\,r \right\} \right. \\ \left[\ \mathit{inoutL} \equiv \lambda\, t\,P\,h \to \mathbf{case}\ t\ \mathbf{of} \left\{ \mathbf{TrI}\,l\,m\,r \to h \right\} \right. \\ \left[\ \mathit{inoutR} \equiv \lambda\, t\,P\,h \to \mathbf{case}\ t\ \mathbf{of} \left\{ \mathbf{TrI}\,l\,m\,r \to h \right\} \right. \end{array} \right\}$$

4 Generic proofs

We can define function that work generically on all *Regular* sets, e.g. a generic equality:

$\mathit{RegEq}\,(R \in \mathit{Regular}\,n\,\mathit{Ps}\,S, \mathit{RV} \in \mathit{RelVec}\,n\,\mathit{Ps}) \in \mathit{Rel}\,S$

…and generically prove that it is substitutive (in a manner similar to that presented in Section 2.5:

$\mathit{RegSubst}(\mathit{SV} \in \mathit{SubstVec}\,n\,\mathit{Ps}\,\mathit{RV}) \in \mathit{isSubstitutive}\,S\,(\mathit{RegEq}\,R\,\mathit{RV})$

In fact, it has been our experience that generic proofs of porperties such as substitutivity and decidability are easier than the non-generic ones!

It is worth noting that our system allows writing generic functions with explicit recursion, rather than recursors. This is consistent with the rest of *Alfa*, which also allows explicit recursion and pattern-matching.

4.1 Towards inductive families

An interesting question is how to encode indexed inductive families, such as

$$\mathit{Vec}\ (A \in \mathit{Set}) \in \mathit{Nat} \to \mathit{Set}$$
$$\mathit{Vec}\ A \equiv \mathbf{data} \left\{ \begin{array}{l} [\,] \in \mathit{Vec}\ A\ \mathbf{Zero} \\ \mathbf{:}\ (m \in \mathit{Nat}, x \in A, \mathit{xs} \in \mathit{Vec}\ A\ m) \in \mathit{Vec}\ A\ (\mathbf{S}\,m) \end{array} \right\}$$

Recall that

$$Nat \approx () + Nat$$

Idea: If R is a regular type, we can represent family $R \to Set$ we can use family operators depending on shape of R

$$Vec\,A \approx \lambda().() \oplus \lambda n.\, Vec\,A\,n$$

where

$$\oplus :: (A \to Set) \to (B \to Set) \to (A + B \to Set)$$

We can construct an universe of shapes of inductive families that are indexed by regular types themselves

$FShape\ (n \in Nat) \in Shape0 \to Set$
$FShape\, n \equiv$

$$\mathbf{data} \begin{Bmatrix} \mathbf{Const}\,(s \in Shape0, c \in Shape\,n) \\ \mathbf{FSum}\,(sl, sr \in Shape0, l \in FShape\ (\mathbf{S}\,n)\ sl, r \in FShape\ (\mathbf{S}\,n)\ sr) \\ \mathbf{FProd}\,(s \in Shape0, l, r \in FShape\,n\,s) \\ \mathbf{FProd2}\,(sl, sr \in Shape0, l \in FShape\,n\,sl, r \in FShape\,n\,sr) \\ \mathbf{FRec} \end{Bmatrix}$$

The Vec type will then have the shape

$fShapeVec \in FShape\ (\mathbf{S}\,\mathbf{0})\ ((\mathbf{()} + \mathbf{Rec})$
$fShapeVec \equiv \mathbf{FSum}\ (\mathbf{Const}\ \mathbf{()})\ (\mathbf{FProd}\ (\mathbf{Const}\ (\mathbf{Par}\ (\mathbf{FS}\,\mathbf{FZ})))\ \mathbf{FRec})$

Inductive families indexed by regular types can be reflected using the $FImage$ function:

$$FImage(n \in Nat, s \in Shape0, P \in Set, S \in Image0\,P\,s \to Set, SV \in SetVec\,n,$$
$$\quad f \in FShape\,n\,s) \in Image0\,P\,s \to Set$$
$$FImage\,n\,s\,P\,S\,SV\ (\mathbf{Const}\,c) \equiv \lambda\,h \to Image\,n\ (S\,h)\ SV\,c$$
$$FImage\,n\,s\,P\,S\,SV\ (\mathbf{FSum}\,l\,r) \equiv$$
$$\quad \lambda\,h \to \mathbf{case}\ h\ \mathbf{of} \begin{Bmatrix} \mathbf{Left}\,a \to FImage\ (\mathbf{S}\,n)\ sl\,P\,\lambda\,h' \to S\,h\ (P\mathbf{:}\ SV)\ l\,a \\ \mathbf{Right}\,b \to FImage\ (\mathbf{S}\,n)\ sr\,P\,\lambda\,h' \to S\,h\ (P\mathbf{:}\ SV)\ r\,b \end{Bmatrix}$$
$$FImage\,n\,s\,P\,S\,SV\ (\mathbf{FProd}\,l\,r) \equiv$$
$$\quad \lambda\,h \to FImage\,n\,s\,P\,S\,SV\,l\,h \wedge FImage\,n\,s\,P\,S\,SV\,r\,h$$
$$FImage\,n\,s\,P\,S\,SV\ (\mathbf{FProd2}\,l\,r) \equiv \lambda h@\langle h_1, h_2\rangle \to$$
$$\quad FImage\,n\,sl\,P\,(\lambda\,h' \to S\,h)\,SV\,l\,h_1 \wedge FImage\,n\,sr\,P\,\lambda\,h' \to S\,h\,SV\,r\,h_2$$
$$FImage\,n\,s\,P\,S\,SV\,\mathbf{FRec} \equiv S$$

4.2 Further work

An interesting question , which is a subject of ongoing research, is how to harness the full power of induction-recursion (as described in [DS99]) to enable generic programming over inductive-recursive types.

References

[AM02] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *WCGP*, 2002. To appear.

[Aug98] Lennart Augustsson. Cayenne — a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP'98)*. ACM Press, September 1998.

[CC99] C. Coquand and T. Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages*, Paris, France, Sep 1999.

[CHJ+01] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[Coq98] Catharina Coquand. The AGDA Proof System Homepage. `http://www.cs.chalmers.se/~catarina/agda/`, 1998.

[DS99] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive and inductive-recursive definitions. In Jean-Yves Girard, editor, *Proceedings 4th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'99, L'Aquila, Italy, 7–9 Apr 1999*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer-Verlag, Berlin, 1999.

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

[HR00] Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 70–84. Springer, 2000.

[Jan00] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Gteborg University, Sweden, May 2000.

[JJ98] P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998. Available from `www.cs.chalmers.se/~patrikj/poly/`.

[Mag94] L. Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 1994.

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

[PR99] H. Pfeifer and H. Rueß. Polytypic proof construction. In Yves Bertot, editor, *Proceedings 12th Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLs'99, Nice, France, 14–17 Sept 1999*, volume 1690 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, Berlin, 1999.