The interpretation is designed to be small and easily understood, rather than efficient. For this reason, the language does not support user declared types for terms, which, though possible, would obscure the implementation. As a consequence, the direct implementation of many terms, e.g. of truth-values and pairs, have assigned types which are more general than desired. To overcome this, I have introduced some additional constants where enforced typing seems desirable. For example, `bot0` instantiates `bot` to take arguments of type $0 =$ `Pol`$[1]$ instead of the more general `Pol`$[X]$. Similarly, to limit the degree of polynomials we instantiate `con` to `conE` : $Y \rightarrow$ `Pol`$[X, Y]$.

8 Further and Related Work

The development of shape polymorphism, and polynomial types is in its infancy. To take but one example, mutually recursive types will require a slightly more sophisticated type system. For example, rose trees with nodes of type A (trees whose nodes have an arbitrary, but finite, number of branches) are given by the initial algebra

$$\mu X.A \times \texttt{Ind}\ [1, X]$$

Here the bound variable is not an indeterminate, so that explicit bounds appear to be required.

In theory, all the operations of interest on inductive types are available, once folding has been captured. In practice, things are complicated by the presence of additional parameters. For example, mapping must be extended from the indeterminates to the cofficients of polynomials and inductive types, which will require links between the data and the functions to be applied. This in turn suggests that polynomial types be ordered, so that the addition of new coefficients increases the size of the type.

Similar considerations can be expected when generalising other standard list combinators, such as `scan`, `zip` and `filter` to novel situations. Best would be to find the means of describing these as syntactic sugar, but even if possible, this will require further generalisations of forms.

While discussing **P2**, Mark Jones pointed out that Gofer can use the type class of functors to support a high-level algorithm for `fold`, whose type is

```
fold : functor f =>
           (f x -> x) -> fix f -> x
```

where `fix f` denotes the initial algebra (e.g. an inductive type) for the functor `f`.

9 Conclusions

Although it has been common knowledge that inductive types are built from polynomials, this fact has not, till now, been used to reveal the underlying shape of polynomial and inductive types in a systematic fashion. **P2** shows that polynomials support a uniform algorithm for folding over inductive types, and a type system that recognises this by giving `fld` a single type.

This work shows that shape polymorphism can be implemented. When developed further, and incorporated into other languages, it promises to eliminate much redundant coding in functional programs.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming*. International Series in Computer Science. Prentice Hall, 1988.

[2] J.R.B. Cockett and T. Fukushima. About **charity**. Technical Report 92/480/18, University of Calgary, 1992.

[3] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Tracts in Theoretical Computer Science. CUP, 1989.

[4] et al Goguen, J.A. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24:68–95, 1977.

[5] P. Hudak and all. Report on the programming language haskell: a non-strict, purely functional language. Technical report, University of Glasgow, 1992. Version 1.2.

[6] C.B. Jay. Shapely types: Exploiting parseability. *Science of Computer Programming*, 1995.

[7] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism. In *Proceedings of the European Symposium on Programming, Edinburgh, 1994*, Lecture Notes in Computer Science. Springer Verlag, 1994.

[8] M. Jones. The implementation of the gofer functional programming system. Technical Report YALEU /DCS/RR-1030, Yale University, 1994.

[9] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Procceding of the 5th ACM Conference on Functional Programming and Compter Architecture*, 1991.

[10] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.

5.1 Evaluation

The rewriting rules are given by the usual β-reduction and `let` rule of the λ-calculus

$$\begin{array}{rcl} (\lambda \mathtt{x.e})\mathtt{a} & \Rightarrow & \mathtt{e[a/x]} \\ \mathtt{let}\ x = a\ \ \mathtt{in}\ \ e & \Rightarrow & \mathtt{e[a/x]}\ . \end{array}$$

together with some δ-rules, associated with the constants:

$$\begin{array}{rcl} \mathtt{cas\ f\ g\ (con\ e)} & \Rightarrow & \mathtt{g\ e} \\ \mathtt{cas\ f\ g\ (var\ p\ e)} & \Rightarrow & \mathtt{f\ p\ e} \\ \mathtt{pmp\ f\ (con\ e)} & \Rightarrow & \mathtt{con\ e} \\ \mathtt{pmp\ f\ (var\ p\ e)} & \Rightarrow & \mathtt{var\ (pmp\ f\ p)\ (f\ e)} \\ \mathtt{fld\ f\ (rec\ p)} & \Rightarrow & \mathtt{f\ (pmp\ (fld\ f)\ p)} \end{array}$$

Of course, evaluation preserves type inference, i.e. if we can infer that $\mathtt{e} : A$ and $\mathtt{e} \Rightarrow \mathtt{e}'$ then we can infer that $\mathtt{e}' : A$. Also, evaluation is type-free, i.e. is independent of the choice of type for a term.

6 Interpretation into System F

System **F** is a type system generated by function types, and universal quantification by type variables [3]. It is powerful enough to define the inductive types, but the arbitrary nesting of quantifiers complicates type-checking enormously.

In this section the typable terms of **P2** will be translated into terms of **F** in a way that preserves non-trivial reductions (i.e. reductions of at least one step). Then the Church-Rosser property and strong normalisation for **P2** will follow from that of **F**.

The only potential source of difficulty is the interpretation of form variables. Perhaps they could be mapped to type variables, but various problems arise with substitutions. Rather than attempt to unravel this knot, observe that every typable term of **P2** has a type in which there are no form variables, obtained by substituting the empty form for them. Now the translation is trivial.

The types without form variables can be interpreted into system **F** by a function $[\![-]\!]$ as follows. Atomic types, type variables and function types are all interpreted as themselves. The polynomial type $\mathtt{Pol}[X, A_0, A_1, \ldots, A_n]$ is interpreted as in (3) by

$$((\ldots([\![A_n]\!]\times[\![X]\!]) \ldots) + [\![A_1]\!])\times[\![X]\!] + [\![A_0]\!]$$

Inductive types are handled just as in **F**. That is, if the interpretation of $\mathtt{Pol}(X : F)$ is P then the inductive type $\mathtt{Ind}\ F$ is interpreted by

$$\Pi X.(P \to X) \to X\ .$$

Universal quantification $\forall X$ of polytypes is translated by ΠX. We will suppress the translation brackets $[\![-]\!]$ as much as possible from now on.

Now the terms of **P2** can be interpreted by terms of **F** in a manner that preserves typing, and maps reductions of **P2** to non-trivial reductions of **F**.

Lemma 6.1 *If $\Gamma \vdash e : A$ then there is a proof in which the only substitutions that occur are applied to the types of combinators and variables.*
Proof By induction on the length of the inference. □

Since the type inference rules of **P2** (other than those for the combinators) are also rules of **F** it follows that we need only provide translations for the combinators.

Let

$$\begin{array}{rcl} P & = & \mathtt{Pol}[X, A_0, A_1, \ldots, A_n] \\ Q & = & \mathtt{Pol}[X, A_1, \ldots, A_n] \\ R & = & \mathtt{Pol}[Y, A_0, A_1, \ldots, A_n] \\ I & = & \mathtt{Ind}[A_0, A_1, \ldots, A_n] \\ S & = & \mathtt{Pol}[I, A_0, A_1, \ldots, A_n] \end{array}$$

where X is a type which may or may not be a variable.

The translation of $\mathtt{con} : A_0 \to P$ is the left inclusion $\iota^1 : A_0 \to A_0 + (Q \times X)$. Similarly,

$$\begin{array}{rcl} \mathtt{var} & \mapsto & \lambda p.\lambda x.\iota^2\langle p, x\rangle \\ \mathtt{cas} & \mapsto & \lambda f^{Q\to X\to Y}.\lambda g^{A_0\to Y}.\lambda p^P.p\ Y\ f'\ g \\ \mathtt{fld} & \mapsto & \lambda f^{P\to X}.\lambda z^I.z\ X\ f \end{array}$$

where f' is the uncurried form of f. The translation of $\mathtt{pmp} : (X \to Y) \to P \to R$ is $\mathtt{pm\ n}$ where

$$\begin{array}{rcl} \mathtt{pm}\ 0 & = & \lambda g^{X\to Y}.\lambda p^P.p \\ \mathtt{pm}\ (n+1) & = & \lambda g^{X\to Y}.\lambda p^P.p\ \mathtt{id}\ ((\mathtt{pm}\ n\ g)\times g) \end{array}$$

and $(f\times g)\langle a, b\rangle = \langle f\ a, g\ b\rangle$. Finally, if `rec` has target type I then its translation is

$$\lambda p^R.\Lambda X.\lambda f^{P\to X}.f(\mathtt{pm}\ n\ (\mathtt{fld}\ f)\ p)$$

The proof that the translation preserves reduction is routine, since all the δ-reductions are mapped to β-reductions. Hence, we have:

Theorem 6.2
Reduction in **P2** *is confluent and strongly normalising.*

7 Implementation

The implementation of **P2** is intended to demonstrate the viability of the ideas in this paper, and provide a vehicle for their further exploration. An interpreter for **P2** has been constructed in Gofer (Jones, [8]) though any functional language would have sufficed, since no special use has been made of its type classes. The code is available by anonymous ftp from ftp.socs.uts.edu.au in the directory users/cbj/P2.

$$\begin{aligned}\sigma[\,] &= [\,]\\ \sigma[Z] &= \sigma_2 Z\\ \sigma(T:K) &= (\sigma_1 T):(\sigma K)\ .\end{aligned}$$

Form substitutions allow the length of a form to change, so that terms whose natural type is a polynomial of low degree can be treated as having a type of high degree.

This novelty is not as radical as it may appear, since it vanishes when the polynomials are converted to the sums of products. For example, if $T = \mathtt{Pol}[X, A, B]$ then two applications of (3) show that

$$\begin{aligned}\mathtt{Pol}[X,Y,T] &\cong (\mathtt{Pol}[X,A,B]\times X) + Y\\ &\cong \mathtt{Pol}[X,Y,A,B]\ .\end{aligned}$$

Hence, substituting T for Z in $\mathtt{Pol}[X,Y,Z]$ is the same (up to isomorphism) as substituting $[A,B]$ for Z in the form. Notice that this equivalence only works when the form variable ends the list.

Substitution on polytypes behaves in the usual way, by not interacting with bound variables, i.e. there is no substitution for bound variables, nor is variable capture allowed.

Composition of substitutions ρ followed by σ is denoted by $\sigma * \rho$. Clearly, the identity substitution maps type variables to themselves, and maps a form variable Z to $[Z]$. The notation $\sigma[T/X]$ is used to express the update of σ where the type (respectively, form) variable X is mapped to the type (respectively, form) T.

4.2 Unification

Unification follows the usual pattern. `Fail` indicates that unification has failed. First strip off bound variables, taking care not to accidentally create new type identifications, and then apply the following algorithm.

$$\begin{aligned}\mathcal{U}(1,1) &= \mathtt{id}\\ \mathcal{U}(X,T) &= \mathtt{id}[T/X]\\ \mathcal{U}(T,X) &= \mathcal{U}(X,T)\\ \mathcal{U}(\mathtt{Pol}\ K, \mathtt{Pol}\ L) &= \mathcal{V}(K,L)\\ \mathcal{U}(\mathtt{Ind}\ K, \mathtt{Ind}\ L) &= \mathcal{V}(K,L)\\ \mathcal{U}(A{\to}B, C{\to}D) &= \sigma_2 * \sigma_1 \text{ where}\\ &\quad \sigma_2 = \mathcal{U}(\sigma_1 B, \sigma_1 D)\\ &\quad \sigma_1 = \mathcal{U}(C,A)\\ \mathcal{U}(A,B) &= \mathtt{Fail}\ \text{(otherwise)}\ .\end{aligned}$$

For $\mathcal{U}(X,T)$ it is assumed that X is not free in T unless $T = X$. Unification on forms is the symmetric operation given by

$$\begin{aligned}\mathcal{V}([\,],[\,]) &= \mathtt{id}\\ \mathcal{V}([Z],[\,]) &= \mathtt{id}[[\,]/Z]\\ \mathcal{V}(K,[\,]) &= \mathtt{Fail}\ \text{(otherwise)}\\ \mathcal{V}(K,[Z]) &= \mathtt{id}[K/Z]\\ \mathcal{V}(T:K, T':K') &= \mathcal{V}(\sigma K, \sigma K') * \sigma\\ &\quad \text{where } \sigma = \mathcal{U}(T,T')\ .\end{aligned}$$

Theorem 4.1 (Most General Unifiers) *If there is a substitution ρ that unifies two types A and B then ρ factors through $\mathcal{U}(A,B)$.*

Proof Without loss of generality, A and B are monotypes. The proof is by a standard induction over both the types and the forms. □

5 Terms

A *context* Γ is a list of term variables with given polytypes. The constant terms and their associated types (in any context) are given by

$$\begin{aligned}! &: 1\\ \mathtt{bot} &: \mathtt{Pol}[X]{\to}Y\\ \mathtt{con} &: Y{\to}\mathtt{Pol}[X,Y,Z]\\ \mathtt{var} &: \mathtt{Pol}[X,Z]{\to}X{\to}\mathtt{Pol}[X,Y,Z]\\ \mathtt{cas} &: (\mathtt{Pol}[X,Z]{\to}X{\to}T){\to}(Y{\to}T){\to}\\ &\quad \mathtt{Pol}[X,Y,Z]{\to}T\\ \mathtt{pmp} &: (X{\to}Y){\to}\mathtt{Pol}[X,Z]{\to}\mathtt{Pol}[Y,Z]\\ \mathtt{rec} &: \mathtt{Pol}[\mathtt{Ind}[Z],Z]{\to}\mathtt{Ind}[Z]\\ \mathtt{fld} &: (\mathtt{Pol}[X,Z]{\to}X){\to}\mathtt{Ind}[Z]{\to}X\ .\end{aligned}$$

The type inference rules for variables, `let`, functions and their application are given by

$$\frac{}{\Gamma \vdash \mathtt{x} : A}\ \mathtt{x} : A \text{ in } \Gamma$$

$$\frac{\Gamma \vdash \mathtt{e} : A \quad \Gamma, \mathtt{x} : A \vdash \mathtt{e}' : B}{\Gamma \vdash \mathtt{let\ x = e\ \ in\ \ e'} : B}$$

$$\frac{\Gamma, \mathtt{x} : A \vdash \mathtt{e} : B}{\Gamma \vdash \lambda\mathtt{x.e} : A{\to}B}$$

$$\frac{\Gamma \vdash \mathtt{f} : A{\to}B \quad \Gamma \vdash \mathtt{e} : A}{\Gamma \vdash \mathtt{f\ e} : B}$$

Finally, there are the rules relating to polytypes and substitutions. Substitutions act on all the types in a context Γ. Here the variable X may be a type or form variable.

$$\frac{\Gamma \vdash \mathtt{e} : A}{\Gamma \vdash \mathtt{e} : \forall X A}\ X \text{ not free in } \Gamma$$

$$\frac{\Gamma \vdash \mathtt{e} : \forall X A}{\Gamma \vdash \mathtt{e} : A[T/X]}$$

$$\frac{\Gamma \vdash \mathtt{e} : T}{\sigma\Gamma \vdash \mathtt{e} : \sigma T}$$

Type assignment proceeds as usual, with unification used to handle application, and polytypes used for `let` constructs.

3 Inductives

Polynomial types can be used to express domain equations for inductive types. For example, the usual equation $N \cong 1 + N$ for the natural numbers becomes

$$N \cong \texttt{Pol}[N, 1, 1] \ .$$

Similarly, lists of type A and binary trees with leaves of type A and nodes of type B are given by

$$\begin{aligned} L &\cong \texttt{Pol}[L, 1, A] \\ T &\cong \texttt{Pol}[T, A, 0, B] \ . \end{aligned}$$

If $PX = \texttt{Pol}[X, A_0, A_1, A_2, \ldots, A_n]$ is a polynomial functor then its initial algebra is the *inductive type*

$$I = \texttt{Ind}[A_0, A_1, A_2, \ldots A_n]$$

which satisfies the domain equation

$$I \cong \texttt{Pol}[I, A_0, A_1, A_2, \ldots A_n] \ .$$

This isomorphism justifies the constructor

$$\texttt{rec} : \texttt{Pol}[\texttt{Ind}[Z], Z] \rightarrow \texttt{Ind}[Z] \qquad (4)$$

where Z is a form variable. It combines with the `pol` notation to give

$$\texttt{ind es} = \texttt{rec(pol es)} \ .$$

For example, the constructors for the type of lists $L = \texttt{Ind}[1, A]$ are given by

$$\begin{aligned} \texttt{nil} &= \texttt{ind}[!] \\ \texttt{cons h t} &= \texttt{ind}[\texttt{h}, \texttt{t}] \ . \end{aligned}$$

Similarly, the constructors for a binary tree $T = \texttt{Ind}[A, 0, B]$ are

$$\begin{aligned} \texttt{leaf a} &= \texttt{ind}[\texttt{a}] \\ \texttt{node(b,t1,t2)} &= \texttt{ind}[\texttt{b}, \texttt{t1}, \texttt{t2}] \ . \end{aligned}$$

The meaning of initiality is that for type T with a P-action

$$\texttt{f} : \texttt{Pol}[T, Z] \rightarrow T$$

there is a (unique) algebra homomorphism

$$\texttt{fld f} : \texttt{Ind}[Z] \rightarrow T$$

whose evaluation is given by

$$\texttt{fld f (ind es)} = \texttt{f (pmp (fld f) (pol es))} \ .$$

That is, `fld f` applies itself to all sub-terms of indeterminate type, and then applies `f`.

Returning to our example of lists, if

$$\texttt{f} = \texttt{cases}[\texttt{c}, \texttt{g}] : \texttt{Pol}\ [C, 1, A] \rightarrow C$$

makes C an algebra for $PX = \texttt{Pol}[X, 1, A]$ then $\texttt{fld f} : L \rightarrow C$ maps lists of A's to C in the usual way:

$$\begin{aligned} \texttt{fld f nil} &\Rightarrow \texttt{c} \ ! \\ \texttt{fld f (ind}[\texttt{h}, \texttt{t}]) &\Rightarrow \texttt{f (pol}[\texttt{h}, \texttt{fld f t}]) \ . \end{aligned}$$

Abstracting over `f` in (5) shows that

$$\texttt{fld} : (\texttt{Pol}[T, Z] \rightarrow T) \rightarrow \texttt{Ind}[Z] \rightarrow T$$

where Z is a form variable. Hence, `fld` has a single type, expressed in terms of polynomial and inductive types, and form variables, whose nature we must now consider in a little more detail.

4 The Type System

The *(mono-)types* are given by

$$\begin{aligned} T &:= 1 \mid X \mid T \rightarrow T \mid \texttt{Pol}\ K \mid \texttt{Ind}\ K \\ K &:= [\,] \mid Z \mid T : K \ . \end{aligned}$$

These types are: terminal (or unit), variable, function, polynomial, and inductive. Other base types could be added, too. K is a *form*. It consists of a list of types which may or not be ended by a *form variable* Z. By convention, X and Y will denote type variables, and Z will denote a form variable, unless otherwise stated.

The *polytypes* include the monotypes and are closed under universal quantification by both type and form variables: if T is a polytype then so is

$$\forall X \ T$$

where X is either a type or a form variable. Polytypes are required for the typing of `let` expresions, where each occurence of a single term may be required to have a different (monotype) instantiation of its polytype.

4.1 Substitutions

A *type substitution* σ is a pair (σ_1, σ_2) of functions which map type variables to types and form variables to forms. Substitution acts on monotypes and forms as follows:

$$\begin{aligned} \sigma C &= C \\ \sigma X &= \sigma_1 X \\ \sigma(T \rightarrow T') &= (\sigma T) \rightarrow (\sigma T') \\ \sigma(\texttt{Pol}\ K) &= \texttt{Pol}\ (\sigma K) \\ \sigma(\texttt{Ind}\ K) &= \texttt{Ind}\ (\sigma K) \end{aligned}$$

and

$$A_0 + A_1 + \ldots + A_n = \texttt{Pol}[1, A_0, A_1, \ldots, A_n]$$

and the *initial type* is the sum of nothing given by

$$0 = \texttt{Pol}[1] \ .$$

Similarly, the type of *booleans* is given by $2 = 1 + 1 = \texttt{Pol}[1, 1, 1]$.

Binary products can be represented by the linear polynomial

$$A \times B = \texttt{Pol}[A, 0, B] \ .$$

Note that, unlike the sums, the n-fold product cannot be described directly, but must be given by repeatedly forming binary products. This might be overcome by allowing multinomial types, which may have several indeterminates.

Polynomials can be described recursively using monomials. That is, $\texttt{Pol}[X, A_0, A_1, \ldots, A_n]$ can be constructed as

$$\texttt{Pol}[X, A_1, \ldots, A_n] \times X + A_0 \qquad (3)$$

which suggests the two constructors of polynomial type: `con`, for constant polynomials, and; `var`, for varying polynomials. We can type them by

$$\begin{aligned} \texttt{con} &: Y \to \texttt{Pol}[X, Y, Z] \\ \texttt{var} &: \texttt{Pol}[X, Z] \to X \to \texttt{Pol}[X, Y, Z] \ . \end{aligned}$$

where Z is a form variable.

For example, the left injection of $\texttt{a} : A$ into the coproduct $A + B$ can be given as `con a`. The right injection of $\texttt{b} : B$ is then `var (con b) !` where $! : 1$ is the canonical term of unit type. Specialising to $A = B = 1$ yields

$$\begin{aligned} \texttt{true} &= \texttt{con !} \\ \texttt{false} &= \texttt{var (con !) !} \ . \end{aligned}$$

Although 0 is an empty type (which denotes the initial object), it does have a canonical function $\texttt{bot} : 0 \to Y$ into any other type Y.

Given a term $\texttt{ek} : A_k$ for $k \leq n$ and terms $\texttt{ei} : X$ for $1 \leq i \leq k$ then we can construct a term of type $\texttt{Pol}[X, A_0, A_1, \ldots, A_n]$ "of degree k" by

```
pol [e0,e1,...,ek] =
      var(...(var(con e0) e1)...) ek
```

That is, `pol (e : es) = pol2 e es` where

$$\begin{aligned} \texttt{pol2 p []} &= \texttt{p} \\ \texttt{pol2 p (e : es)} &= \texttt{pol2 (var p e) es} \ . \end{aligned}$$

The corresponding selector `cas` for polynomials performs a case analysis in the obvious way. Its type is

$$(\texttt{Pol}[X, Z] \to X \to T) \to (Y \to T) \to \texttt{Pol}[X, Y, Z] \to T$$

and its evaluation rules are:

$$\begin{aligned} \texttt{(cas f g) (var p e)} &\Rightarrow \texttt{f p e} \\ \texttt{(cas f g)(con e)} &\Rightarrow \texttt{g e} \ . \end{aligned}$$

The extension to multiple cases is handled as follows. Given $\texttt{fs} = [\texttt{f0}, \texttt{f1}, ..., \texttt{fn}]$ where `fk` is a function of type

$$\texttt{fk} \ : \ A_k \to X \to X \ldots \to X \to T \ (k \text{ copies of } X)$$

for $0 \leq k \leq n$ then `cases fs` has type

$$\texttt{Pol}[X, A_0, A_1, \ldots, A_n] \to T \ .$$

More precisely, `cases` is defined by

$$\begin{aligned} \texttt{cases []} &= \texttt{bot} \\ \texttt{cases (f : fs)} &= \texttt{cas (cases fs) f} \ . \end{aligned}$$

The use of `bot` for the empty list places an upper bound on the degree of the terms to which the case analysis applies. For the recursion, observe that $\texttt{cases}[\texttt{f1}, \ldots, \texttt{fn}]$ has type

$$\texttt{Pol}[X, A_1, \ldots, A_n] \to (X \to T)$$

since each `fk` has had its position in the list reduced by one. This is just right for typing

$$\texttt{cas (cases}[\texttt{f1}, \ldots, \texttt{fn}]\texttt{) f0} \ .$$

Anyway, it follows that

```
cases fs (pol [e0,e1,...,ek]) =>
   fk e0 e1 ... ek .
```

Similarly, if $\texttt{fk} \ : \ A_k \to T$ for $0 \leq k \leq n$ then we can construct the usual case analysis on the sum of the A_k by ignoring the copies of 1 that arise.

$$\begin{aligned} \texttt{scs []} &= \texttt{bot} \\ \texttt{scs (f : fs)} &= \texttt{cas (fst.(scs fs)) f} \end{aligned}$$

where `fst x y = x`. In particular, the conditional $\texttt{cond} : \texttt{Pol}[A, 0, 2] \to A$ is given by

$$\texttt{cond b f g} \ = \ \texttt{cases}[\texttt{bot}, \texttt{bot}, \texttt{scs}[\texttt{pi2}, \texttt{pi3}]]$$

where `pi2 x y z = y` and `pi3 x y z = z`.

Our final polynomial combinator maps functions over the indeterminates of polynomials. It has type

$$\texttt{pmp} : (X \to Y) \to \texttt{Pol}[X, Z] \to \texttt{Pol}[Y, Z]$$

with evaluation given by

$$\begin{aligned} \texttt{pmp f (var p e)} &\Rightarrow \texttt{var (pmp f p) (f e)} \\ \texttt{pmp f (con e)} &\Rightarrow \texttt{con e} \ . \end{aligned}$$

The recursion in the evaluation shows why `pmp` cannot be defined in terms of `cas`. It follows from the definition that

```
pmp g (pol [e0,e1,...,en]) =
           pol [e0,g e1,...,g en]
```

In this paper, folding over arbitrary inductive types will be characterised by its own combinator, with its own type, and a single evaluation mechanism. Thus, the uniform description of the inductive types is reflected in a uniform description of their algorithms.

To see how this is done, we must reformulate the problem. The pattern-matching approach to `lsum` and `tsum` is required to explain where the recursion is to occur, either on the tail of the list, or the left and right sub-trees. In other words, if we can locate the "recursive sub-expressions" then we can describe `fold f` by the following high-level algorithm:

> Apply `fold f` to all the recursive sub-expressions and then apply `f`.

Hence the task is to construct a type system which supports the location of the recursive sub-expressions.

A clue to the solution is provided by the *shapely types* (Jay and Cockett, [7] and Jay, [6]). They support the separation of *data* (e.g. the values at the leaves of a tree) from *shape* (e.g. the underlying, unlabelled tree), which can then be manipulated separately. For example, an operation whose shape is fixed but allows the data to vary is *data polymorphic*, of the kind found in most functional programming languages. Conversely, if the data is fixed but the shape may vary then it is *shape polymorphic*, a new kind of polymorphism which we are currently investigating. The canonical example of a shape polymorphic function is `map`, which applies a function to each datum, but leaves the shape fixed.

This clue is not quite enough to solve the problem, however, since this absolute separation of data from shape does not match the recursive approach to folding. The two views are combined in the *polynomial types*, since they express their shapeliness in a suitably recursive fashion.

Polynomial types are isomorphic to types built from products and sums. For example,

$$\texttt{Pol}[X, A_0, A_1, A_2] \cong A_0 + A_1 \times X + A_2 \times X^2 \ .$$

They are more expressive, however, since the ability to distinguish the *indeterminate type* from the *coefficient types* provides just the desired shape information: recursion occurs at the indeterminate types.

For example, (1) and (2) can be recast as

$$\begin{aligned} L &\cong \texttt{Pol}[L, 1, A] \\ T &\cong \texttt{Pol}[T, A, 0, 1] \ . \end{aligned}$$

Thus each polynomial yields a *polynomial functor*

$$PX = \texttt{Pol}[X, A_0, A_1, A_2, \ldots, A_n] \ .$$

whose corresponding initial algebra is the inductive type

$$\texttt{Ind}[A_0, A_1, A_2, \ldots, A_n]$$

sometimes denoted $\mu X.PX$. In this case there is no need to specify the bound variable X since it is always the indeterminate of the polynomial.

The use of a term of polynomial type supports a single, general algorithm for evaluating `fold f`. The list of coefficients of the polynomial, its *form*, carries both the data (the actual types of the coefficients) and the shape (the length of the list). By introducing form variables, folding can be represented by a constant term or combinator `fld`, instead of a term constructor. That is, `fld` is not merely data polymorphic but is shape polymorphic, too.

In this paper, a language **P2** of polynomial types and terms is introduced, together with type inference, unification and assignment algorithms. Evaluation is shown to be Church-Rosser and strongly normalising.

Polynomial and inductive types are introduced in Sections 2 and 3. The formal system of types and terms is given in Sections 4 and 5. In Section 6 the terms are interpreted within System **F**, from which the Church-Rosser property and strong normalisation follow. The paper concludes with comments on implementation, further work, related work and conclusions in Sections 7–9.

I would like to thank the members of the **Shape** project and the referees for their comments.

2 Polynomials

A *polynomial type*

$$\texttt{Pol}[X, A_0, A_1, A_2, \ldots, A_n]$$

is built from a *form* $[X, A_0, A_1, A_2, \ldots, A_n]$ whose *indeterminate* is the type X and whose *coefficient* of *degree* k is the type A_k. There are also form variables Z used to represent lists of unknown coefficients. They will be addressed in detail in Section 4.

In a system which supports product and sum types the polynomial above may be interpreted by the type

$$A_0 + A_1 \times X + A_2 \times X^2 + \ldots + A_n \times X^n \ .$$

Notice, however, that the distinction between indeterminates and coefficients has been lost.

Here, we take the polynomial types, and a unit (or terminal) type 1 to be primitive and construct other types from them. Sums are represented by

$$A + B = \texttt{Pol}[1, A, B] \ .$$

More generally, finite sums are given by

Polynomial Polymorphism

C. Barry Jay

School of Computing Sciences
University of Technology, Sydney
Sydney
Australia
cbj@socs.uts.edu.au.

Abstract

Inductive types, such as lists and trees, have a uniform semantic description, both of the types themselves and the folding algorithms that construct homomorphisms out of them. Though implementations have been able to give a uniform description of the types, this has not been true of folding, since there has not been a uniform mechanism for finding the sub-expressions (the sub-lists or sub-trees, etc.) to which recursion applies.

Polynomial types overcome this problem by distinguishing the indeterminate of the polynomial (on which the recursion occurs) from its coefficients. Further, this uniformity is recognised by the type system, which is able to treat `fld` *as a (shape) polymorphic constant of the* λ*-calculus.*

These ideas have been implemented in a language **P2**.

Key words types, polynomials, polymorphism, folding, shape, P2.

1 Introduction

The use of initial algebras to describe inductive data types, such as lists and trees, produces a uniform method for describing a large class of types (ADJ, [4]) and specifying their programs (e.g. Meijer et al, [9]). Till now, however, this uniformity has not extended to the algorithms used for implementation.

In languages such as ML [10] and Haskell [5], many algorithms are defined by pattern-matching. For example, summation of a list of integers is given by

$$\begin{aligned} \texttt{lsum []} &= \texttt{0} \\ \texttt{lsum (h :: t)} &= \texttt{h} + \texttt{(lsum t)} \end{aligned}$$

while summation over a binary tree of integers is given by

$$\begin{aligned} \texttt{tsum(leaf(n))} &= \texttt{n} \\ \texttt{tsum(node(t1,t2))} &= \texttt{(tsumt1)} + \texttt{(tsum t2)} \ . \end{aligned}$$

That is, while these languages support polymorphism in the choice of *data*, operations must be separately defined for each choice of *shape* (lists, trees, etc.).

Semantically, inductive data types can be treated as initial algebra solutions $F^\dagger$ of domain equations $X \cong FX$. An algebra for F is a type B with an F*-action* $f : FB \to B$. For each such there is a unique algebra homomorphism (or *catamorphism*) $\texttt{fold}\ f : F^\dagger \to B$.

For example, lists of A's and binary trees with leaves of type A are solutions of the equations

$$L \cong 1 + A \times L \quad (1)$$

$$T \cong A + T^2 \quad (2)$$

that are initial algebras for the functors $GX = 1 + A \times X$ and $HX = A + X^2$ respectively. An algebra B for the list functor F is given by some $f : (1 + A \times B) \to B$ or, equivalently by a chosen element $b : B$ and an A-action $a : A \to B \to B$. For example, $\texttt{lsum} = \texttt{fold}\ g$ where g is given by 0 and $+$.

Likewise, an algebra B for the tree functor H is given by a function $A \to B$ and a binary operation $B \to B \to B$. For example $\texttt{tsum} = \texttt{fold}\ h$ where h is given by the identity and $+$.

The Bird-Meertens formalism (Bird and Wadler, [1]) supports a fold over lists called `foldr` whose type is

$$(A \to B \to B) \to B \to [A] \to B \ .$$

so that $\texttt{lsum} = \texttt{foldr}\ +\ \texttt{0}$. Similar combinators could be introduced for other inductive types, each with their own evaluation mechanism, but their underlying unity would not be revealed this way.

The language **Charity** (Cockett and Fukushima, [2]) takes a different approach to the problem. There `fold` applies to arbitrary inductive types, but is treated as a term constructor, rather than as a term in its own right. That is, an algorithm for `fold g` is inferred from that of `g`. Thus, the use of folding has been simplified by automating its construction, but the algorithm itself is still dependent on that for `g`.