Polytypic Recursion Patterns

L. S. Barbosa, [a] J. B. Barros, [a] J. J. Almeida [a]

[a] *Computer Science Department, University of Minho, Portugal*
{lsb,jbb,jj}@di.uminho.pt

Abstract

Recursive schemes over inductive data structures have been recognized as category-theoretic *universals*, yielding a handful of equational laws for program construction and transformation. This paper introduces the implementation of such recursion patterns as type parametric, or *polytypic*, functionals in the Camila prototyping language. Several examples are discussed.

1 Introduction

Polytypic or *generic* programming [8,10] deals with algorithmic constructions that are defined uniformly over a (large) class of data types and, therefore, abstracted with respect to their type constructors. In fact, a polytypic program distinguishes itself from a polymorphic one in that the parameter is a type constructor (*i.e.*, a map, like "tree" or "sequence", from types to types) rather than a type (*e.g.*, an "integer" or a "tree of strings"). The potential advantage of genericity is that it makes possible to write programs to deal with an entire class of problems once and for all, instead of writing new code for each different instance. Besides this increased potential for reuse, generic programs get stripped of irrelevant detail, becoming eventually more reliable and easier to reason about.

The notions of a *functor* and of its *algebra*, borrowed from category theory, provide the right level of abstraction to derive polytypic programs. Recall that a category is basically a space of similar structures — the *objects* — and structure-preserving transformations — the *arrows*. Categories may therefore act as representations of particular computation models by taking types as objects and operations upon them as arrows. In such a setting, giving a type constructor amounts to specify both a way of building new types from old

⋆ Partially supported by LogComp (PRAXIS XXI - 2/2.1/TIT/1658/95).

and of lifting to the new types the operations defined on their components. In functional programs over sequences, the later is known as the `map` functional. These two aspects are joined together in the notion of a functor, *i.e.*, a transformation of categories acting uniformly upon objects and arrows.

Is there a common principle underlying standard recursive programs over, *e.g.*, natural numbers, sequences or binary trees? A reasonable answer would stress the relationship between the recursion pattern and the structure of the data involved. There are, of course, different recursion patterns, with varying degrees of applicability, but all of them can be recognized as depending on the *shape* of the data structure that the algorithm consumes, generates or (as discussed in section 6) simply "rests on". Therefore they can be incorporated on real programming languages as *polytypic* functionals. This is the point of the present paper. Five common kinds of recursive patterns, abstracted over the type constructors, are primitively incorporated in CAMILA. Suitable instantiations are automatically generated for each user-defined datatype.

CAMILA [1] is an experimental platform for formal software development, providing a functional prototyping kernel and a refinement calculus [15] for *model-oriented* specifications [11]. If there is a slogan characterizing this approach to software engineering, it will state something like "specify the data models and the rest will be given". Our contribution amounts to include recursion patterns in such a "rest". The paper should be read as a "proof of concept", in the sense that it reports on a practical implementation of recent and challenging developments. The CAMILA basic datatypes and the definition mechanism are introduced in the next section. Then, section 3 discusses canonical recursive schemes over inductive data types and the way they are incorporated in the language. Section 4 provides a non trivial case-study — the Davis-Putman procedure for testing validity of propositional formulae. Some work in progress in reported on section 5.

The categorical view of datatypes, which underlies this research area, dates back to the ADJ group [5], and more recently, to the contributions of T. Hagino [7] and G. Malcolm [12]. The relevance of universal properties to program derivation was first recognized by Backhouse in [3]. References [14,13] and [16] introduce the recursion functionals discussed here. Reference [4] provides a tutorial introduction.

2 Data Modeling in CAMILA

CAMILA has been designed as a language for rapid prototyping of model-oriented specifications. and resembles a centenary notation of *naive* set theory, something we are used to regard as a *tool to think with*. In fact, the CAMILA prototyping environment is just a *calculator* for a fragment of set theory. For most practical purposes one can take as a semantic universe for

CAMILA specifications the category Set of sets and set-theoretic functions. However this is not exactly so: an order-enriched category is needed as long as one wants to deal with partiality and guarantee unique solutions to type equations [1]. Datatypes are specified as combinations of regular functors and the direct powerset functor $\mathcal{P}\_$, over the semantic category. Therefore, as basic constructors CAMILA provides *Cartesian product* ($A \times B$), *sum* ($A + B$) and *exponential* (A^B), to express aggregation, choice and functional dependence, respectively.

Product

The cartesian product of X and Y is the set $X \times Y = \{(x, y) |\ x \in X \wedge y \in Y\}$ together with the projection morphisms $\pi_1 : X \times Y \longrightarrow X$ and $\pi_2 : X \times Y \longrightarrow Y$ defined by $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$. The *product* type in CAMILA is declared as `X * Y`, with `p1`, `p2`, ..., as projections. Being a functor, it may also be applied to functions $f : X \longrightarrow X1$ and $g : Y \longrightarrow Y1$, yielding $f \times g : X \times Y \longrightarrow X1 \times Y1$ such that $f \times g = \lambda\,(x, y)\,.\,(f\,x, g\,y)$. In CAMILA this is written as `prod(f,g)` or `(f _x_ g)` [2].

The universal arrow associated to products is *splitting*. Given $f : W \longrightarrow X$ and $g : W \longrightarrow Y$ the *split* of f and g is the function $\langle f, g\rangle : W \longrightarrow X \times Y$ defined as $\langle f, g\rangle = \lambda\,w\,.\,(f\,w, g\,w)$. The concrete CAMILA syntax is `split(f,g)`.

The language also provides a *record* type, isomorphic to an n-ary product, but with projections identified by user-defined labels. The concrete syntax is `L1:X L2:Y`, where `L1` and `L2` are (optional) labels. When processing type declarations, CAMILA generates automatically not only such *selectors*, but also a *constructor* with the name of the type, as well as *bridging* functions to witness the isomorphism between a record type and the corresponding simple product, *e.g.*

$$\texttt{T = L1:X L2:Y} \underset{\texttt{in-T}}{\overset{\texttt{out-T}}{\rightleftarrows}} \texttt{X * Y}$$

In practice such functions are quite useful as they allow to take a record type value, convert it to a simple tuple, apply standard functions (such as `prod` or `split`) and build again the result as a record.

---

[1] A suitable candidate is `Cpo`, the category of pointed complete partial orders, imposing a definedness order on each type. This is the default universe for functional languages semantics (see *e.g.* [4]), although, as a category, it lacks some structure. For example sums fail to be coproducts and moreover the corresponding universal property holds only for strict functions. In the sequel, however, we will not bother excessively about such semantic considerations.

[2] Its generalization to n-ary products is `sprod(l)`, `l` being a sequence of functions. Such generalizations are valid for all constructors to be introduced in the sequel. The rule is to append the prefix `s` to the operator name and supply the argument as a sequence of suitably typed functions. So one ends up with an `ssum`, an `ssplit`, an `seither`, and so on.

Sum

Dualising the product construction one gets the *sum* (or disjoint union) of two types. This is defined, in Set, as $X+Y = (\{1\}\times X)\cup(\{2\}\times Y)$, together with the embedding morphisms $i_1 : X \longrightarrow X+Y$ and $i_2 : Y \longrightarrow X+Y$ verifying $i_1(x) = (1,x)$ and $i_2(y) = (2,y)$. The corresponding CAMILA syntax is `X + Y`, with the embeddings i_1 and i_2 written as `i1` and `i2`, respectively. CAMILA also provides *origin-identifier* predicates `is1` and `is2`, widely used *e.g.* in the VDM metalanguage [11]. As a (bi-)functor its action on functions is given by $f+g : X+Y \longrightarrow X_1+Y_1$ such that

$$f+g = \lambda\, t\,.\, \begin{cases} i_1(f\,x) & \text{if } t = (1,x) \\ i_2(g\,x) & \text{if } t = (2,x) \end{cases}$$

In CAMILA this construction is written as `sum(f,g)` or `(f _+_ g)`. The universal associated to a sum [3] is the *either* construction: given $f : X \longrightarrow W$ and $g : Y \longrightarrow W$, $[f,g] : X+Y \longrightarrow W$ is defined as

$$[f,g] = \lambda\, t\,.\, \begin{cases} f\,x & \text{if } t = (1,x) \\ g\,x & \text{if } t = (2,x) \end{cases}$$

and written in CAMILA as `either(f,g)`.

To achieve compatibility with other formal methods notations, CAMILA also offers a more verbose version of sums in which the embeddings are explicitly labelled by the component types, instead of being so positionally. The resulting type is called an *alternative* and is written as `T = X | Y` . The origin-identifier predicates `is-X` and `is-Y` are automatically generated from the type declaration [4]. So are the *bridging* functions (`out-T` and `in-T`) to convert to and from simple sums. The notation `[X]` is used for the case $X+\mathbf{1}$, where $\mathbf{1}$ stands for the singleton set (represented in CAMILA by `ONE`, whose canonical inhabitant is the constant `NIL`), due to its heavy use in modeling exception situations and design partiality.

Exponential

The *exponential* type Y^X, declared as `X --> Y`, corresponds to the space of operations (total functions in Set, continuous functions in Cpo) from X to Y. Therefore, functions are first class citizens in CAMILA. In particular, they can be given as arguments or returned as a result to other functions. Moreover they may appear in the definition of other types, for example to model a record type in which some components are themselves operations. Standard constructions on a function space include: *composition* (`f _o_ g`), *identity* (`id`, defined as $\lambda\, x\,.\, x$), *curry* (`curry`, the well-known isomorphism $C^{A\times B} \longrightarrow C^{B^A}$) and *constants* (`_c v`, denoting $\underline{v} \triangleq \lambda\, x\,.\, v$, for a value v).

---

[3] but be aware of the restriction mentioned in footnote 1.

[4] The summands may carry an optional label, as in `T = L:X | K:Y` , generating `is-L` and `is-K`.

Derived Constructors
The kernel language also includes the following derived constructors, for finite A and B: $\mathcal{P}A$ (*finite subsets*), $A \rightharpoonup C$ (*finite mappings*), defined as $\sum_{K \subseteq A} C^K$, and A^* (*finite sequences*), defined as $\sum_{n \subseteq \mathbb{N}} A^n$, *i.e.*, the initial solution of the polynomial functor $\mathsf{F}X = \mathbf{1} + A \times X$.

By functoriality, all those constructs act upon functions (either primitive or user-defined) lifting its effect to the generated structure. For example, the expressions `(f-set)-seq` and `(f-seq)-set` correspond, respectively, to the action upon the function $f : A \longrightarrow B$ of the functors $\mathcal{P}\_^*$ and $\mathcal{P}\_^*$.

3 Generic Recursive Functionals

Functors and Algebras
Under the slogan *type constructors are functors*, let us consider now the definition of recursive types in Camila. A *sequence*, for example, is either empty or consists of a distinguished element (the *head*) and another sequence (the *tail*). A binary tree may also be empty or aggregate information into the root node and two subtrees. A useful variant stores effective information only on the leaves. And so on. Notice that all those examples can be modeled by a polynomial functor, which is basically a n-ary sum (of alternatives) of m-ary products (of information associated to each alternative). In particular

$$\begin{aligned}
\mathsf{F}_{\mathsf{Nat}}\, X &= \mathbf{1} + X && \textit{natural numbers}\\
\mathsf{F}_{\mathsf{Seq}}\, X &= \mathbf{1} + A \times X && \textit{sequences}\\
\mathsf{F}_{\mathsf{Bin}}\, X &= \mathbf{1} + A \times X \times X && \textit{binary trees}\\
\mathsf{F}_{\mathsf{Lef}}\, X &= A + X \times X && \textit{leaf trees}
\end{aligned}$$

Natural numbers and sequences are primitively defined in Camila, but the other cases have to be declared together with explicitly named constructors. As it will become clear below, those are necessary to allow for the automatic generation of the recursion functionals. For example, the datatype of *leaf trees*, corresponding to $\mathsf{F}_{\mathsf{Lef}}$ and polymorphic on A, is declared as

```
Ltree  = lLtree | nLtree;
lLtree = leaf: ANY;
nLtree = left: Ltree  right: Ltree;
```

From such a declaration Camila generates, as explained above, the constructors `lLtree` and `nLtree`, and the projections `leaf`, `left` and `right`. Moreover it also generates the *functorial extension* of `Ltree`, named `map-Ltree` [5].

The inductive type T arises, in each case, as the solution to the equation $X \cong \mathsf{F}\,X$. This isomorphism is witnessed by a bijection whose components are $in_{\mathsf{F}} : \mathsf{F}\,T \longrightarrow T$ and $out_{\mathsf{F}} : T \longrightarrow \mathsf{F}\,T$. The former specifies how values of

[5] The functorial extension of a type `F` is always denoted by `map-F`.

T are built from a set of constructors. For example $in_{\mathsf{F}_{Seq}}$ is the *either* of the sequence constructs nil : $\mathbf{1} \longrightarrow T$ and cons : $A \times X \longrightarrow T$, written in CAMILA as `either(_c <>, cons)`. Dually, the latter describes how values of T can be *observed*. Again, for the $\mathsf{F}_{\mathsf{Seq}}$ example, this is written in CAMILA as

```
out-Seq <- (_c <> _+_ split(hd,tl)) _o_ grd(eqk(<>));
```

where `eqk(<>)` is the predicate `lambda(l). l == <>` and `grd` transforms a predicate p on a set A into a guard $p? : A \longrightarrow A + A$, separating the values of A for which p holds from the others it does not. In the general case, for an arbitrary datatype `F`, the system generates both the functionals `in-F` and `out-F`. For example, for $\mathsf{F}_{\mathsf{Lef}}$ we get

```
in-Ltree  <- either(lLtree, nLtree);
out-Ltree <- (leaf _+_ split(left, right)) _o_ grd(is-lLtree);
```

or, in a more verbose, pointwise notation,

```
out-Ltree <- lambda(t). if (is-lLtree(t) -> i1(leaf(t)),
                            is-nLtree(t) -> i2(<left(t), right(t)>));
```

Catamorphisms

The values of T are terms and, in fact, the pair $\langle T, in_{\mathsf{F}_{Seq}} \rangle$ is the *term-algebra* for the F_{Seq} functor. Again the designation is borrowed from category theory, where the algebra for a functor is simply an arrow from $\mathsf{F}\,X$ to an object X, called the *carrier*. This simple notion is extremely expressive. In particular, there is a one to one correspondence between algebras for certain Set-functors and the usual notion of an algebra in Universal Algebra. The term-algebra corresponds exactly to the one which is *initial* among all such arrows. Being initial means that there is exactly a unique arrow from its carrier to any other algebra, *i.e.* the function h represented by a dotted arrow in the following diagram. For this example T is well known to be isomorphic to A^*.

$$\begin{array}{ccc} C & \xleftarrow{\;f\;} & \mathbf{1} + A \times C \\ {\scriptstyle h}\,\vdots\uparrow & & \uparrow{\scriptstyle \mathsf{id}+\mathsf{id}\times h} \\ T & \xleftarrow[\;in_{\mathsf{F}_{Seq}}\;]{} & \mathbf{1} + A \times T \end{array}$$

It is the existence and uniqueness of such an arrow h that makes possible definition and proof by induction, respectively. That is actually the reason why such datatypes are classified as *inductive*. Observe now that h corresponds to what has been known in functional programming as a *fold* operator: the algebra f specifies an algorithmic step, while $\mathsf{F}\,h$ takes care of recurring and seeking termination. To see this instantiate the parameter A with the natural numbers and define f as $[\underline{0}, +]$. The diagram expresses the equation

$$h \cdot [\underline{\mathrm{nil}}, \mathrm{cons}] \;=\; [\underline{0}, +] \cdot (\mathsf{id} + \mathsf{id} \times h)$$

or, going pointwise,

$$\begin{aligned} h\ \mathrm{nil} &= 0 \\ h\ \mathrm{cons}(x, l) &= x + h\, l \end{aligned}$$

In general, by choosing a different monoid for f, one gets the corresponding monoidal reduction. Notice that h is totally determined by the datatype shape, F, and the one step function f. Following [14], h is written as $(\!|f|\!)_{\mathsf{F}}$ and called the F-*catamorphism* on f. The diagram for the general case is

$$\begin{array}{ccc} C & \xleftarrow{\ f\ } & \mathsf{F}\, C \\ {\scriptstyle (\!|f|\!)_{\mathsf{F}}}\uparrow & & \uparrow{\scriptstyle \mathsf{F}\,(\!|f|\!)_{\mathsf{F}}} \\ T & \xleftarrow[\ in_{\mathsf{F}}\ ]{} & \mathsf{F}\, T \end{array}$$

whose commutativity expresses the following universal property:

$$h = (\!|f|\!)_{\mathsf{F}} \iff h \cdot in_{\mathsf{F}} = f \cdot \mathsf{F}\, h$$

The construction $(\!|\ |\!)$ is a *polytypic* functional, as it is parametric on the type constructor F. Moreover it codifies a structural recursion pattern entirely determined by the underlying data structure. This functional is generated in Camila as `cata-F`.

Paramorphisms

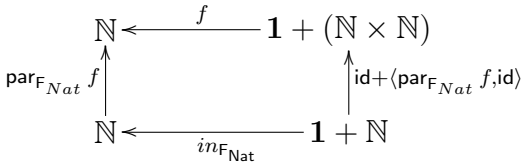
In several cases the result of a recursive function may depend not only on computations in substructures of its argument, but also on the substructures themselves. When applied to the particular case of the inductive definition of the natural numbers (*cf.*, $\mathsf{F}_{\mathsf{Nat}}$), such a pattern is known as *primitive recursion*. In the general case, this can be captured by considering the application of F to the product of the result with the inductive type as the source of f. This smooth generalization of the cata recursion pattern is due to [13] and known as a *paramorphism*. It is defined in the following diagram

$$\begin{array}{ccc} C & \xleftarrow{\ f\ } & \mathsf{F}\,(C \times T) \\ {\scriptstyle \mathsf{par}_{\mathsf{F}}\, f}\uparrow & & \uparrow{\scriptstyle \mathsf{F}\,\langle \mathsf{par}_{\mathsf{F}}\, f, \mathsf{id}\rangle} \\ T & \xleftarrow[\ in_{\mathsf{F}}\ ]{} & \mathsf{F}\, T \end{array}$$

which entails the following universal property

$$h = \mathsf{par}_{\mathsf{F}}\, f \iff h \cdot in_{\mathsf{F}} = f \cdot \mathsf{F}\,\langle h, \mathsf{id}\rangle$$

The usual factorial function arises by suitable instantiation of the diagram:

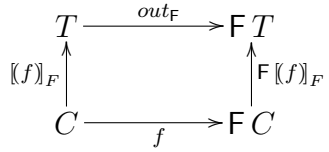


This is specified in CAMILA as

```
fac <- para-Nat(facstep);
facstep <- either(_c 1, (mul _o_ (id _x_ succ)));
```

Anamorphisms

A new recursion functional is obtained by reversing the arrows in the catamorphism diagram. Technically this means we turned attention to *coalgebras* for F, *i.e.*, arrows from T to $\mathsf{F}\,T$. As our working category is order-enriched, type equations have unique fix points and this makes T also the carrier of the *final* coalgebra. And, again, this entails a universal property: there is a unique arrow from any other coalgebra to $\langle T, out_{\mathsf{F}} \rangle$ respecting the functor structure, *i.e.*, making the following diagram commute. Such unique arrow is called an F-*anamorphism* and written, for a given f, as $[\!(f)\!]_F$, or, in CAMILA, `ana-F(f)`,



entailing

$$h = [\!(f)\!]_F \iff out_{\mathsf{F}} \cdot h = \mathsf{F}\,h \cdot f$$

As an example take again $\mathsf{F}_{\mathsf{Seq}}$ and choose as a target coalgebra the function $apstep : A^* \times A^* \longrightarrow \mathbf{1} + A \times (A^* \times A^*)$ defined in CAMILA as

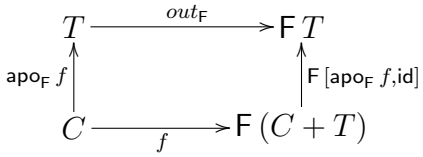
```
apstep(k,l) = if ((k == <> /\ l == <>) -> i1(NIL),
                  (k == <> /\ l != <>) -> i2(<hd(l), <k, tl(l)>>),
                  (k != <>)            -> i2(<hd(k), <tl(k), l>>));
```

This function specifies an iteration step. Its anamorphism $[\!(\texttt{apstep})\!]_F$ — written in CAMILA as `ana-Seq(apstep)` — computes the concatenation of a pair of lists, polymorphic on A. The recursion pattern in order is now *unfolding*: the result is generated from new *seed* values progressively produced in each iteration. The function $[\!(\texttt{apstep})\!]_F$ is often said to be defined coinductively [6].

[6] The expressive power of anamorphisms can only be clearly appreciated in categories, such as Set, in which type equations do have different fix points. In that case the least fix point corresponds to the inductive type whereas the greatest one becomes the carrier of the

Apomorphisms

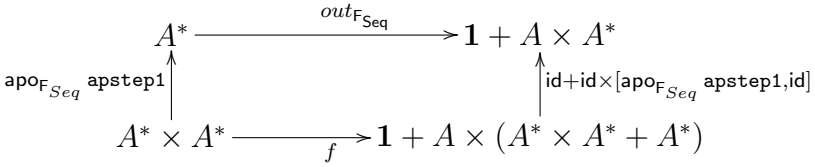
The concatenation algorithm above does not look particularly efficient as both arguments are copied to the resulting sequence. A better solution is provided by immediately returning the second argument as soon as the first list gets exhausted. This is expressed by a slightly more general functional, which is the formal dual to the paramorphism pattern discussed above. The new functional is called an *apomorphism* (first introduced in [16]) and allows for the final result to be either generated in successive steps or "all at once" without recursion. Therefore, the codomain of f becomes $C + T$, instead of simply C. The diagram is



which entails the following universal property

$$h = \mathsf{apo}_{\mathsf{F}}\, f \iff out_{\mathsf{F}} \cdot h = \mathsf{F}\,[h, \mathsf{id}] \cdot f$$

The usual algorithm for list concatenation arises then as $\mathsf{apo}_{\mathsf{F}_{Seq}}\,\mathtt{apstep1}$ in



In CAMILA one writes

```
apseq1 <- apo-Seq(apstep1);
apstep1(k,l) = if ((k == <> /\ l == <>) -> i1(NIL),
                   (k == <> /\ l != <>) -> i2(<hd(l), i2(tl(l))>),
                   (k != <>)            -> i2(<hd(k), i1(<tl(k), l>)>));
```

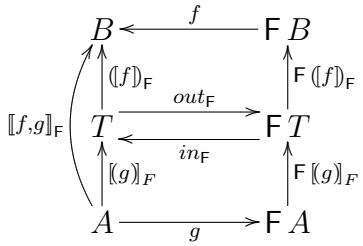
Hylomorphism

The last recursion pattern, implemented as a functional in CAMILA has a surprisingly wide application in practice. In fact, it abstracts the common algorithmic principle "*unfold and then fold*", *i.e.*, unfold the argument to populate an intermediate data structure and, then, fold over such structure to build the intended result. This functional, known as a *hylomorphism* [14],

---

coinductive type, usually expressing non finite constructions. For example the coinductive type associated to $\mathsf{F_{Seq}}$ is the set of finite and infinite sequences, a structure closer to the representation of a *behavior* than of a proper data structure.

is depicted in the following diagram and corresponds to the composition of a cata with an anamorphism,



which entails the following universal property

$$h = [\![f, g]\!]_{\mathsf{F}} \iff f \cdot \mathsf{F}\, h \cdot g = h$$

However the sequencing of an unfold and fold phases is replaced by a single monolithic recursion which does not explicitly construct the intermediate data structure. Also notice that the correctness of the definition depends crucially on the uniqueness of fix points for the type equations. This has been the main point in favor of moving from Set to an order enriched setting as a semantic universe for functional languages.

The kernel of algorithms following this recursion pattern do lie on the intermediate — actually virtual! — structure. Let us illustrate this point with two more algorithms to compute the factorial function written in Camila. Both of them begin by unfolding its argument n into an auxiliary structure populated with its n predecessors. Then such collection is reduced to the result value. The algorithms differ exactly on the intermediate structure: a *sequence* in the first case, a *leaf tree* in the second. We end up with a single or a double recursive algorithm, reflecting this crucial choice.

To give an hylomorphism in Camila amounts to specify the source coalgebra and the target algebra, respectively `fa` and `fc`, and `dfa` and `dfc`, in the examples below. Then those functions are supplied to the hylo functional corresponding to the functor characterizing the intermediate data structure. Following are the codifications of the two versions of the factorial function. The reader is invited to draw the corresponding diagrams.

```
fa <- either(_c 1, mul);
fc <- (_c nil _+_ split(id, pred)) _o_ grd(eqk(0));

fac1 <- hylo-Seq(fa,fc);
```

and

```
dfa <- either(id, mul);
dfc(p) =
  let (n = p1(p), m = p2(p)) in
  if  ( (n == m) -> i1(n),
```

```
    (n != m) -> let (t = div(m .+ n, 2) )
                in  i2(<<n,t>, <succ(t), m>>)
  );
```

In this last case the hylomorphism is defined for $n > 0$ and the terminal case supplied separately, *i.e.*,

```
fac2 <- lambda(n). if (n==0 -> 1, else -> hylo-Ltree(dfa,dfc)(<1,n>));
```

It is surprising the number of problems that can be modeled by hylomorphisms [7]. Moreover, from a specification point of view, the interest of hylomorphisms lies in their potential to *classify* algorithms apparently unrelated. For example it becomes not only instructive, but also useful, from a software engineering point of view, to find out that, *e.g.*, the Fibonacci function and the double factorial or, on the other hand, quicksort and towers of hanoi, do belong to the same families. The first two are $\mathsf{F}_{\mathsf{Lef}}$-hylomorphisms, the other two hylos for $\mathsf{F}_{\mathsf{Bin}}$.

4 A Case Study

This section shows how a non trivial algorithm for testing the validity of propositional formulae may be described in a very concise way as an hylomorphism, directly implemented in Camila. Some variants are considered afterwards.

The Davis-Putman Procedure is a classical algorithm to test propositional validity. The following short description is based on [6]. Let Φ be a proposition. Then,

(i) Test whether Φ belongs to a class of particularly simple propositions, whose validity test can be performed in some trivial way; in this case the procedure stops, yielding the value of this simple test.
(ii) Choose a propositional symbol p (from Φ) and compute
  (a) Φ_+ – a proposition equivalent to Φ assuming that p holds;
  (b) Φ_- – a proposition equivalent to Φ assuming that $\neg p$ holds.
(iii) Apply the same process to both Φ_+ and Φ_-. The value to be returned is the conjunction of these two partial results.

A Camila *Implementation*
We begin with the declaration of the datatype `Prop` of propositional formulae, which is hopefully self-explanatory.

```
Prop =  T: ONE | F: ONE | Not: Prop | And: Prop * Prop | Or: Prop * Prop;
```

[7] See, for example, [2] for a systematic presentation of sorting algorithms as hylos on different inductive types.

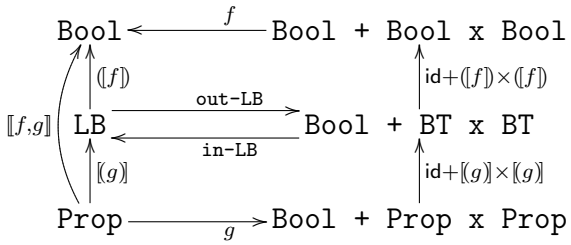
We may now try to capture the algorithm description directly by the following function in CAMILA.

```
davisPutman(phi) =
   if (simple(phi) -> simpleT(phi),
       else        -> let (p = propSy(phi),
                           <phiplus, phiminus> = divide(p,phi))
                      in davisPutman(phiplus) /\ davisPutman(phiminus));
```

However, this definition can be both *explained* by, and *rewritten* as, a hylomorphism over tree-like structure `LB`, a leaf tree with boolean leaves:

```
LB     =  LBleaf | LBnode;
LBleaf =  lb: Bool;
LBnode =  sy: ANY  nl: LBtree rl: LBtree;
```

The hylomorphism is depicted in the following diagram



its *genes*, f and g, being defined as

```
f <- either(id, and);
g <- lambda(phi) . if (simple(phi) -> i1(simpleT(phi)),
                       else        -> i2(divide(propSy(phi),phi)));
```

Therefore, the `davisPutman` function is rewritten as

```
davisPutman <- hylo-LB(f,g);
```

Let us take a look at the intermediate structure generated for a simple case [8]. Let Φ be the proposition

$$(r \vee \neg q) \wedge (p \vee q \vee r) \wedge (\neg p \vee r) \wedge (q \vee s \vee \neg p) \tag{1}$$

Let r be the propositional symbol used to split Φ; the propositions Φ_+ and Φ_- are respectively, $(q \vee s \vee \neg p)$, and $\neg q \wedge (p \vee q) \wedge (\neg p) \wedge (q \vee s \vee \neg p)$

- Let p be the propositional symbol used to split Φ_+; the propositions Φ_{++} and Φ_{+-} are respectively, True, and $(q \vee s)$.
  - The first of these (Φ_{++}) falls under that class of simple cases; it will therefore give rise to a leaf of the tree.

[8] Conjunctive/disjunctive normal forms are particularly well-suited to perform the computation of Φ_+ and ϕ_-.

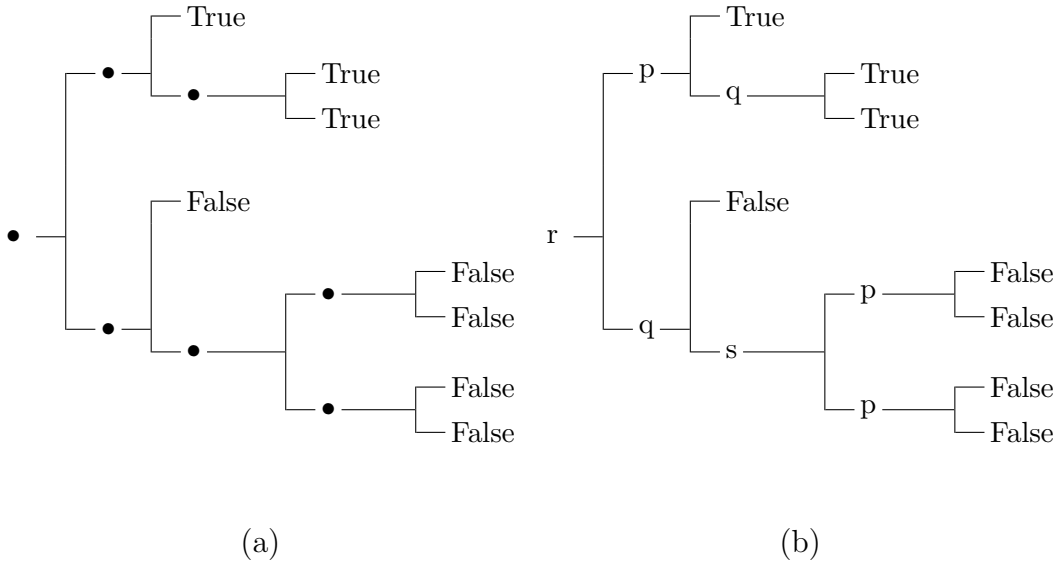


Fig. 1. Davis Putman Procedure

  - Let q be the propositional symbol used to split Φ_{+-}; the propositions Φ_{+-+} and Φ_{+--} are both True, thus giving rise to two leaves of the tree.
- Let q be the propositional symbol used to split $\Phi-$; the propositions Φ_{-+} and Φ_{--} are, respectively, False, and $p \wedge (\neg p) \wedge (s \vee \neg p)$
  - The first of these will result in a leaf of the tree
  - Let s be the propositional symbol used to split $\Phi--$; the propositions Φ_{--+} and Φ_{---} are, respectively, $p \wedge \neg p$, and $p \wedge \neg p \wedge \neg p$

    Let p be the propositional symbol used to split Φ_{--+}; the propositions Φ_{--++} and Φ_{--+-} are both False.

    Let p be the propositional symbol used to split Φ_{---}; the propositions Φ_{---+} and Φ_{----} are both False.

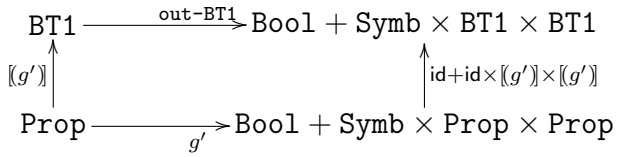
The tree generated by this execution is (as shown by the indentation above) dipicted in Figure 1 (a).

Variants

Note that the intermediate nodes of the tree produced by the Davis-Putman procedure do not carry any kind of real information. Suppose that, instead of generating this tree, we decide to fill in the intermediate nodes with the propositional symbols used to make the various branches. The tree would then look like a binary decision tree to test for the validity of a proposition (see Figure 1 (b)).

In order to produce this tree, a similar generation algorithm is used, but determined by a slightly different data structure, as may be seen from the

following diagram



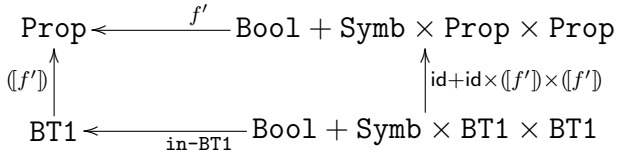
Quite immediately one gets

```
formtree <- ana-BT1(g');
```

The interesting point is that the *gene* of this anamorphism over `BT1` is very similar to the previous one. In fact,

```
g' <- lambda(phi) . if (simple(phi) ->  i1(simpleT(phi)),
                        else        ->  let (p = propSy(phi))
                                        in  i2(<p,divide(p,phi)>));
```

Not surprisingly, from this decision tree one can recover a proposition which is equivalent to the original one. All we have to do is to write a suitable catamorphism filling the diagram



i.e.,

```
recover <- cata-LB1(f');
```

with

```
f' <- either(rAtom, rForm);

rAtom <- lambda(b). if (b -> T,  else -> F);
rForm <- lambda(t). let (p = p1(t), t1 = p1(p2(t)), t2 = p2(p2(t)) )
                    in And(Or(rAtom(p),t1), Or(Not(rAtom(p)),t2));
```

Pasting this diagram on top of the previous one, we get the composite

```
formtree _o_ recover
```

which corresponds to the hylomorphism

```
simplify <- hylo-LB1(f',g');
```

which can be used as a *simplification procedure* for propositional formulae. Notice, however, that the structure of `f'` is rather poor and, consequently, the simplification achieved is minimal. By defining more sophisticated ways of performing the catamorphism part, one may get a more effective simplification procedure. A possible candidate is

```
f'' <- either(rAtom, rForm');
```

```
rForm' <- lambda(t).
          let (p = p1(t), t1 = p1(p2(t)), t2 = p2(p2(t)) )
          in  if (is-T(t1)
                    -> if (is-T(t2) -> t1,
                           is-F(t2) -> rAtom(p),
                           else     -> Or(rAtom(p),t2)),
                  is-F(t1)
                    -> if (is-T(t2) -> Not(rAtom(p)),
                           is-F(t2) -> t1,
                           else     -> And(Not(rAtom(p)),t2)),
                  else
                    -> if (is-T(t2) -> Or(Not(rAtom(p)),t1),
                           is-F(t2) -> Or(rAtom(p),t1),
                           else     -> And(Or(rAtom(p),t1), Or(Not(rAtom(p)),t2)))
                 );
```

yielding

```
simplify-indeed <- hylo-LB1(f'',g');
```

which, for example, simplifies proposition (1) to $r \wedge q \wedge s \wedge \neg p$.

5 Conclusions and Further Work

In summary, from an (inductive) type declaration, the CAMILA interpreter generates a *kit* of functionals which act as the building blocks for user-defined operations. It does so by instantiating polytypic versions of such functionals. The *kit* includes the *functorial action* (`map-F`), *initial algebra* (`in-F`) and its *inverse* (`out-F`), as well as the recursion functionals just described (`cata-F`, `para-F`, `ana-F`, `apo-F` and `hylo-F`). Work in progress includes

- the design of a library of polytypic functions for CAMILA, in the spirit of [9], but completely based in the recursion patterns described here, and their *classification* by the datatypes involved.
- The articulation of these constructors with the *embedding* facilities of CAMILA to create (polytypic) hybrid, component-based, prototypes.

The interest of the recursion patterns discussed in this paper lies not only their polytypic character, but also on the possibility they offer of writing structured definitions of recursive functions *without* making explicit the recursive calls. In fact, programming exclusively in terms of generic functionals directly derived from datatype definitions, such as catamorphisms or anamorphisms, leads to a controlled, *data driven*, use of recursion. This may be as beneficial to declarative programming as the removing of `goto` statements has been to imperative languages twenty years ago.

References

[1] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Prototyping and refinement of constructive specifications. In M. Johnson, editor, *6th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 554–559, Sydney, December 1997. Springer Lect. Notes Comp. Sci. (1349).

[2] L. Augusteijn. Sorting morphisms. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming, Braga*, pages 1–27. Springer Lect. Notes Comp. Sci. (1608), September 1998.

[3] R. Backhouse. An exploration of the Bird-Meertens formalism. CS 8810, Groningen University, 1988.

[4] R. Bird and O. Moor. *The Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997.

[5] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *Jour. of the ACM*, 24(1):68–95, January 1977.

[6] J. Goubault-Larrecq and I. Mackie. *Proof Theory and Automated Deduction.* Kluwer Academic Publishers, 1997.

[7] T. Hagino. *Category Theoretic Approach to Data Types.* Ph.D. thesis, tech. rep. ECS-LFCS-87-38, Laboratory for Foundations of Computer Science, University of Edinburgh, UK, 1987.

[8] P. F. Hoogendijk. *A generic theory of datatypes.* Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology, 1996.

[9] P. Jansson and J. Jeuring. POLYP - a polytypic programming language extension. In *POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[10] J. Jeuring and P. Jansson. Polytypic programming. In T. Launchbury, E. Meijer, and T. Sheard, editors, *International Summer School on Advanced Functional Programming*, pages 68–114. Springer Lect. Notes Comp. Sci. (1129), 1996.

[11] Cliff B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason (IFIP), editor, *Information Processing 83*, pages 321–332. Elsevier Science Publishers B. V. (North-Holland), 1983.

[12] G. R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

[13] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[14] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Lect. Notes Comp. Sci. (523), 1991.

[15] J. N. Oliveira. Software reification using the SETS calculus. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).

[16] V. Vene and T. Uustalu. Functional programming with apomorphisms (corecursion). In *Proc. 9th Nordic Workshop on Programming Theory*, 1997.