Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire

Erik Meijer * Maarten Fokkinga † Ross Paterson ‡

Abstract

We develop a calculus for lazy functional programming based on recursion operators associated with data type definitions. For these operators we derive various algebraic laws that are useful in deriving and manipulating programs. We shall show that all example functions in Bird and Wadler's "Introduction to Functional Programming" can be expressed using these operators.

1 Introduction

Among the many styles and methodologies for the construction of computer programs the Squiggol style in our opinion deserves attention from the functional programming community. The overall goal of Squiggol is to *calculate* programs from their specification in the way a mathematician calculates solutions to differential equations, or uses arithmetic to solve numerical problems.

It is not hard to state, prove and use laws for well-known operations such as addition, multiplication and —at the function level— composition. It is, however, quite hard to state, prove and use laws for arbitrarily recursively defined functions, mainly because it is difficult to refer to the recursion scheme in isolation. The algorithmic structure is obscured by using unstructured recursive definitions. We crack this problem by treating various recursion schemes as separate higher order functions, giving each a notation of its own independent of the ingredients with which it constitutes a recursively defined function.

This philosophy is similar in spirit to the 'structured programming' methodology for imperative programming. The use of arbitrary goto's is abandoned in favour of structured control flow primitives such as conditionals and while-loops that replace fixed patterns of goto's, so that reasoning about programs becomes feasible and sometimes even elegant. For functional programs the question is which recursion schemes are to be chosen as a basis for a calculus of programs. We shall consider several recursion operators that are naturally associated with algebraic type definitions. A number of general theorems are proven about these operators and subsequently used to transform programs and prove their correctness.

Bird and Meertens [4, 18] have identified several laws for specific data types (most notably *finite* lists) using which they calculated solutions to various programming problems. By embedding the calculus into a categorical framework, Bird and Meertens' work on lists can be extended to arbitrary, inductively defined data types [17, 12]. Recently the group of Backhouse [1] has extended the calculus to a relational framework, thus covering indeterminancy.

*University of Nijmegen, Department of Informatics, Toernooiveld 6525 ED Nijmegen, e-mail: erik@cs.kun.nl
†CWI, Amsterdam & University of Twente
‡Imperial College, London

Independently, Paterson [21] has developed a calculus of functional programs similar in contents but very dissimilar in appearance (like many Australian animals) to the work referred to above. Actually if one pricks through the syntactic differences the laws derived by Paterson are the same and in some cases slightly more general than those developped by the Squiggolers.

This paper gives an extension of the theory to the context of lazy functional programming, i.e., for us a type is an ω-cpo and we consider only continuous functions between types (categorically, we are working in the category CPO). Working in the category SET as done by for example Malcolm [17] or Hagino [14] means that finite data types (defined as initial algebras) and infinite data types (defined as final co-algebras) constitute two different worlds. In that case it is not possible to define functions by induction (catamorphisms) that are applicable to both finite and infinite data types, and arbitrary recursive definitions are not allowed. Working in CPO has the advantage that the carriers of initial algebras and final co-algebras coincide, thus there is a single data type that comprises both finite and infinite elements. The price to be paid however is that partiality of both functions and values becomes unavoidable.

2 The data type of lists

We shall illustrate the recursion patterns of interest by means of the specific data type of cons-lists. So, the definitions given here are actually specific instances of those given in §4. Modern functional languages allow the definition of cons-lists over some type A by putting:

$$A* \ ::= \ \mathrm{Nil} \mid \mathrm{Cons}\ (A \| A*)$$

The recursive structure of this definition is employed when writing functions $\in A* \to B$ that destruct a list; these have been called *catamorphisms* (from the greek preposition $\kappa\alpha\tau\alpha$ meaning "downwards" as in "catastrophe"). *Anamorphisms* are functions $\in B \to A*$ (from the greek preposition $\alpha\nu\alpha$ meaning "upwards" as in "anabolism") that generate a list of type $A*$ from a seed from B. Functions of type $A \to B$ whose call-tree has the shape of a cons-list are called *hylomorphisms* (from the Aristotelian philosophy that form and matter are one, $\upsilon\lambda o$ meaning "dust" or "matter").

Catamorphisms

Let $b \in B$ and $\oplus \in A \| B \to B$, then a list-catamorphism $h \in A* \to B$ is a function of the following form:

$$\begin{aligned} h\ \mathrm{Nil} &= b \\ h\ (\mathrm{Cons}\ (a, as)) &= a \oplus (h\ as) \end{aligned} \tag{1}$$

In the notation of Bird&Wadler [5] one would write $h = \mathrm{foldr}\ b\ (\oplus)$. We write catamorphisms by wrapping the relevant constituents between so called banana brackets:

$$h \ = \ (\!|b, \oplus|\!) \tag{2}$$

Countless list processing functions are readily recognizable as catamorphisms, for example $\mathrm{length} \in A* \to \mathrm{Num}$, or $\mathrm{filter}\ p \in A* \to A*$, with $p \in A \to \mathrm{bool}$.

$$\begin{aligned} \mathrm{length} &= (\!|0, \oplus|\!) \ \textit{where}\ a \oplus n = 1 + n \\ \mathrm{filter}\ p &= (\!|\mathrm{Nil}, \oplus|\!) \\ &\quad \textit{where}\ a \oplus as = \mathrm{Cons}\ (a, as), \quad p\ a \\ &\qquad\qquad\ = as, \qquad\qquad\ \neg p\ a \end{aligned}$$