Review of Algebra of Programming Richard Bird and Oege de Moor Prentice Hall, 1996 (dated 1997).

Background

The histories of functional programming and program transformation have been intertwined from their inception. Serious program manipulations are not feasible in modern imperative languages which allow aliasing, pointer and reference modifications, type casting and so forth. More suited are current functional languages which support the definition of general operations – such as map, filter and fold over lists – as polymorphic higher-order functions. The properties of these functions – such as $map\,(f \circ g) = map\,f \circ map\,g$ – can be expressed in a logic which extends the definitional equality of the programming language, and largely equational reasoning in that logic allows transformations to be written down in a formal way.

Early work in this field is typified by Rod Burstall and John Darlington's fold/unfold transformations (see, for instance, (Darlington, 1982)) in which new – recursive – definitions of functions are built using the defining equations left-to-right (unfolding) and right-to-left (folding).

Similarly influential is the work of John Backus and his collaborators on the language FP (Backus, 1978). Backus's response to the well-publicised 'software crisis' was to design a combinator-based language in which functions were to be given explicit, variable-free definitions. These definitions would be more amenable to being manipulated formally, and indeed a literature of FP transformations emerged. On the other hand, the relatively limited uptake of FP showed that users of functional languages were unhappy to eschew the more free-wheeling definition forms of the mainstream functional languages emerging at that time.

During the late 1970s and 1980s a systematic corpus of functional transformations was built up by many researchers, but perhaps most prominently by Richard Bird and Lambert Meertens. Notable in their work is the formalisation of the theory of lists (Bird, 1978), and related theories of bags and sets, brought together in so-called Boom hierarchy.

In the 1990s several people, including Bird and de Moor, have worked at generalising this theory of program transformations from functions to relations. The obvious advantage of a relations over functions is in increased expressiveness, which makes it possible to reason about specifications, non-deterministic and partial functions.

The book presents the state of the art in this approach. The first three chapters introduce an algebra of functions, the next three chapters extend this theory from functions to relations, and the final four chapters apply this theory to various optimisation problems.

Category theory

The traditional foundation for mathematics is the language and theory of sets: functions, relations and indeed all mathematical structures can be reduced to sets, but in practice it is often of little help to look at the underlying set representation. An alternative, higher-level, approach is given by category theory. Instead of representing structures as sets, a category consists of the structure-preserving functions ('arrows') between the structures, together with the structures ('objects') themselves. From a slightly different point of view, then, a category is an abstract theory of functions, as was known to Joachim Lambek and Dana Scott in their independent work on the relationship between the typed lambda-calculus and the theory of cartesian closed categories in the 1970s.

The categorical approach is *point-free*, so that constructions – such as the product of two types A and B – which are traditionally defined by giving their elements – the pairs (a, b) for a in A and b in B – are instead axiomatised by describing the functions between them and related types. The categorical definition of product states that

- there are projections $\mathit{fst} : A \times B \to A$ and $\mathit{snd} : A \times B \to B$
- for every two functions $f : C \to A$ and $g : C \to B$ there is a unique function $\langle f, g\rangle : C \to A \times B$ so that the equations

  $$f = \mathit{fst} \circ \langle f, g\rangle \qquad\qquad g = \mathit{snd} \circ \langle f, g\rangle$$

  hold. This is often abbreviated by saying that $\langle f, g\rangle$ is the unique function making the diagram

  $$\begin{tikzcd} & C \arrow[dl, "f"'] \arrow[d, "\langle f, g\rangle"] \arrow[dr, "g"] & \\ A & A \times B \arrow[l, "\mathit{fst}"] \arrow[r, "\mathit{snd}"'] & B \end{tikzcd}$$

  commute.

Transforming functional programs

Category theory, then, can give a general account of various programming phenomena, and in the field of program transformation it has been used successfully to describe general forms of definition over inductively defined ('algebraic') data types, the theory of catamorphisms, anamorphisms, and the like. This theory is covered in Chapters 1–3 of Bird and de Moor's book, in which they introduce the relevant concepts of category theory while simultaneously looking at their applications in functional programming and transformation.

Their categorical approach means that they are able to provide general rules for equational program manipulation. Prominent among these is the Fusion Law which states that

$$h \circ (\!| f |\!) = (\!| g |\!)$$

under the circumstance that $h \circ f = g \circ Fh$. The 'banana' brackets denote a catamorphism, that is a generalisation of foldr over lists, and so the rule gives a situation in which a composition including a fold can be made into a fold itself. Moreover, the law applies to all algebraic (or initial) data types, so that there is no need separately to develop theories for lists, binary trees, rose trees and so on.

Does this generality have a cost? We have to examine the foundations of the approach to see the limitations of this algebra of functional programming.

First, the approach is total – no partial objects are available. In itself, this is an advantage rather than a disadvantage. It means that there are no awkward 'undefined' elements around, which invalidate many program transformations. The only possible drawback is the restriction in expressiveness it brings, in particular the restriction to the recursion scheme of catamorphisms. Primitive recursion is a simple generalisation of the catamorphism, and can be coded up using catamorphisms; other more general schemes can also be supported. (Recent work – see for instance (Telford and Turner, 1997) – shows that it is possible for total systems to support complex recursion schemes.) However, there are algorithms that require more liberal forms of recursion than catamorphisms (e.g. fast exponentiation).

A system of combinators is functionally complete relative to a programming language if all terms definable in the language are definable using the combinators; in other words, applications of the combining forms provide the mechanism to simulate the definitions of the full language. The system of categorical combinators given by Bird and de Moor is functionally complete relative to the simply typed lambda calculus (with definitions by primitive recursion over structured types) so that the reader is able to code his or her own definitions in the categorical language used for program manipulation. This language is, of course, a proper subset of Haskell or ML because of the limited forms of recursion available.

The system of categorical combinators used by Bird and de Moor has one main disadvantage and one main advantage.

The disadvantage is that even simple programs like factorial requires some manipulation to be given a catamorphic form, and a two argument function like concat requires substantial machinery to put it in catamorphic form, and thus make it amenable to manipulation. Similar problems require a detour into distributive categories to handle the conditional form of the filter function. It is arguable that this weight of categorical overhead makes their work less approachable than it might be.

A pleasant feature of the categorical approach is the degree to which their reasoning can be equational. In particular the McCarthy conditional form (which is the function-level equivalent of the if ... then ... else ...) gives case-free reasoning for functions for which a more traditional approach would require proof by cases.

Relations

The first three chapters of the book can be seen as a preamble to the remainder, where the calculus of functions is replaced by a calculus of relations, based on the generalisation of a category to an *allegory*, as discussed in Freyd and Scedrov (1990). Taking an instrumental point of view, what does this additional generality bring by way of benefits and specifically what extra tools does it give the programmer?

First, the increased power of expressivity that relations provide in comparison with functions is useful for specifications. For instance, every relation has an inverse, and so it is possible formally to specify a (complicated) function — such as a parser, or a text formatter – by saying it is the inverse of a simpler function, namely a pretty printer or un-formatter. Since the calculus is relational, the (relational) inverse always exists, and can be manipulated within the laws provided by the system. Another example of an operation on relations useful for specifying is the relational meet, the intersection of two relations. This can be used to specify a program incrementally, by conjoining two constraints into a single one; the laws of the calculus allow users to manipulate expressions which involve meets. A further advantage is that the system supports reasoning about containment of one relation in another, so permitting a proof of equality of two relations by showing that each is contained in the other.

Another advantage of relations over functions is that the relations include the *non-deterministic* functions. This means that an algebra of relational programming allows us to reason about non-deterministic functions. This is of vital importance in the optimisation problems discussed in the final chapters.

Finally, the relations also include the *partial* functions. So an algebra of relational programming provides all the familiar partial functions, but without the complexity of partial elements. In particular, it provides the fixpoint operator without the complexity of partial elements and divergence normally associated with it: a non-well-founded recursion will not produce a non-terminating function, but rather an empty relation. Note that this means that one disadvantage of the functional approach, namely the restriction to recursive functions that are catamorphisms, no longer applies in the relation setting.

Bird and de Moor's approach is to develop the appropriate theory by reference to the category of sets and the relations between them. They develop a system of relational combinators with the advantage and disadvantage of this combinatorial approach discussed earlier. While in the functional context the reader is always able to refer to their intuition about functions definable using variables, since the various sets of combinators are known to be complete, relative to the simply typed lambda calculus with primitive recursion, this is by no means so clear here. In the bibliographical remarks to Chapter 4 the authors observe that 'the axioms introduced here ... precisely constitute the definition of a topos'. It would be interesting to develop this theme further, and to see what language of relations (akin to the lambda calculus) is equivalent to their combinators, and also to see the precise relationship between this system and a more traditional relational calculus a la Tarski and adherents of his approach.

Applications

The theory developed in the first six chapters is put into practice in the final four chapters which look at various optimisation problems. The authors point out that the application of techniques like dynamic programming and greedy algorithms go beyond the traditional areas of combinatorial optimisation. Among the applications are that of splitting text into paragraphs, in which they compare a greedy algorithm (as appears in most word processing systems) with an optimal solution as implemented in TeX. The approach of the authors is to find the appropriate generic description of a particular form of program definition which are then applied across a range of fields.

Overall

The book is very clearly written, and its approach of introducing categorical concepts only as they are needed works well for the most part. It is only in looking at the theory of allegories that it becomes difficult to see 'the big picture' of what underlies their approach. As was said earlier in this review, it would be helpful to be able to see the particular set of relational combinators as a realisation of a higher-level theory; maybe this is the place for a more thorough discussion of topos theory and its internal logic?

As mentioned earlier, there are many advantages in moving from a algebra of functions to an algebra of relations. However, there is a price to pay for this that cannot be avoided, namely the inherent complexity of the calculus of relations. This means there is a lot of theory to go through before the interesting applications can be given in the last four chapters.

The book is full of exercises of an appropriate level, and hints for solutions are available electronically. Particularly valuable are the bibliographical remarks which conclude each chapter and which provide evidence of the complex historical background to the work in relational algebra, category theory, theory of algorithms and program transformation.

It has become a commonplace in computing science to distinguish between formality and rigour. Bird and de Moor's book is written with a mathematician's rigour, and so can move successfully between different levels of abstraction when that is necessary. It begs the question of how easily the ideas can be formalised and implemented in a working system of program transformation. The authors have taken a big step in making rigorous some powerful and fundamental ideas in programming. It remains to others to take these ideas and make them a part of a practical computer programming system, but there is every expectation that this will happen and that this will in turn make the authors' ideas accessible to a wider audience.

The text under review provided the theme of the Theoretical Computer Science seminar at the University of Kent for a considerable part of 1997; we are indebted to participants in the seminar, and particularly to Eerke Boiten, for their insights into the material of this challenging work.

References

John Backus. (1978). Can programming be liberated from the Von Neumann style? *Communications of the ACM*, **21**(8).

Richard Bird. (1987). An introduction to the theory of lists. In M. Broy (ed), *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag.

John Darlington. (1982). Program transformation. In J. Darlington, P. Henderson and D. A. Turner (eds.), *Functional programming and its applications*. Cambridge University Press.

Peter Freyd and Andre Scedrov. (1990). *Categories, Allegories*. North-Holland.

Alastair Telford and David Turner. (1997). Ensuring Streams Flow. In M. Johnson (ed), *Algebraic methodology and software technology, 6th international conference, AMAST'97, Sydney Australia*, pp. 509–523. Lecture Notes in Computer Science vol. 1349. Springer-Verlag.