Algorithmics

Richard Bird[1], Jeremy Gibbons[1], Ralf Hinze[2], Peter Höfner[3], Johan Jeuring[4], Lambert Meertens[1], Bernhard Möller[5], Carroll Morgan[6](✉), Tom Schrijvers[7], Wouter Swierstra[4], and Nicolas Wu[8]

[1] University of Oxford, Oxford, UK
{bird,jeremy.gibbons}@cs.ox.ac.uk, lambert@kestrel.edu
[2] Technische Universität Kaiserslautern, Kaiserslautern, Germany
ralf-hinze@cs.uni-kl.de
[3] Australian National University, Canberra, Australia
peter.hoefner@anu.edu.au
[4] Utrecht University, Utrecht, Netherlands
{j.t.jeuring,w.s.swierstra}@uu.nl
[5] Universität Augsburg, Augsburg, Germany
bernhard.moeller@informatik.uni-augsburg.de
[6] University of New South Wales and Data61 (CSIRO), Sydney, Australia
carroll.morgan@unsw.edu.au
[7] KU Lueven, Lueven, Belgium
tom.schrijvers@cs.kuleuven.be
[8] Imperial College London, London, England
n.wu@imperial.ac.uk

Abstract. *Algorithmics* is the study and practice of taking a high-level description of a program's purpose and, from it, producing an executable program of acceptable efficiency. Each step in that process is justified by rigorous, careful reasoning at the moment it is taken; and the repertoire of steps allowed by that rigour, at each stage, guides the development of the algorithm itself.

IFIP's Working Group 2.1 [i] has always been concerned with Algorithmics: both the design of its notations and the laws that enable its calculations. ALGOL 60 had already shown that orthogonality, simplicity and rigour in a programming language improves the quality of its programs.

Our Group's title "Algorithmic Languages and Calculi" describes our activities: the discovery of precise but more general rules of calculational reasoning for the many new styles of programming that have developed over the 60 years since IFIP's founding. As our contribution to the birthday celebrations, we outline how we have tried to contribute during those decades to the rigorous and reliable design of computer programs of all kinds—to *Algorithmics*. (Roman-numbered references like [i] in this abstract refer to details given in Sect. 10.)

Keywords: Working groups · Algorithmic programming · Calculi

1 Introduction

WG2.1 is one of the the first Working Groups of IFIP, and the oldest extant: it was founded at the request of TC2, which had begun its own very first meeting only two days before [ii]. Initially the "IFIP Working Group 2.1 on ALGOL", it is now known as the

IFIP Working Group 2.1 on Algorithmic Languages and Calculi. [iii]

The Group has always focused on methods for systematic program construction; and our goal is to make the methods steadily more powerful and more general. For example, the formalisation of the inductive assertion method [iv] led to a logical method based on pre- and postconditions [v], and then to a strongly calculational goal-directed method [vi]. Generalising programs to special cases of specifications [vii] led to the *Mathematics of Program Construction*. And a program-algebraic approach evolved from that: the "Laws of Programming" [viii].

Mathematics (of program construction or otherwise) can be carried out with pencil and paper. For programs, however, there are more significant advantages in automation than for mathematics generally; thus the Group has always paid attention to program transformation systems [ix]—but their design should be based on the 'by hand' calculations that preceded them.

Language design, including the advancement of ALGOL, remained a main interest for many years, focussing for a period specifically on a more advanced language called "Abstracto". Abstracto generalised what 'programming' languages actually *should* be: rather than just for programming or writing executable code, they should also be able to describe algorithms in an abstract way. They should allow expressing (initially vague) ideas about an algorithm's high-level structure and, after transformations adding details, reach a level from which the final step to 'real' programming-language code is simple enough to minimise the risk of transcription errors. In sum, Abstracto was supposed to support and codify our *Algorithmics* activity: but our activity itself outgrew that.

ALGOL 60 and 68 were languages more oriented to programmers' thoughts than to computers' hardware. In their 'successor' Abstracto, we wanted [xi]

> ...a programming language some of whose features we know:
>
> 1. It is very high level, whatever that means. (1)
> 2. It is suitable for expressing initial thoughts on construction of a program.
> 3. It need not be (and probably is not) executable...

Abstracto was to be an *algorithmic language*: one for describing the algorithmic steps in a computation, not just the input-output relation or similar behavioural specification. But it was still intended to be a 'tool of thought', rather than primarily an implementation language.

But the Abstracto approach was *itself* soon abstracted by abandoning the imperative ALGOL-like language structures, switching to a more functional presentation [xii] in which there was an algebra of programs *themselves*, rather than

say an algebra of statements *about* programs. The framework for this became known as the "Bird–Meertens Formalism", a very concise notation in which algorithmic strategies can be expressed and transformed (Sect. 2). That exposed many general algorithmic patterns and calculational laws about them that had, until then, been obscured by the earlier imperative control structures.

A similar abstracting approach was applied to *data* structures in the form of a hierarchy –the Boom hierarchy– leading from sets through multisets (bags) and lists to (binary) trees [xiii] (Subsect. 2.3, Sect. 3). The insight was that all these structures had a common pattern of constructors (an empty structure, a singleton constructor, and a binary combiner). They were distinguished from each other not by the signatures of their operations, but rather by the algebraic laws imposed on the constructors: the fewer laws, the more structure in the generated elements.

A further abstraction was to allow the constructors to vary, i.e. to have an even more general approach in which one could say rigorously "Sum the integers in a structure, no matter what its shape." and then reason effectively about it, for example that "Square all the integers in a structure, and then add them up." is the same as "Sum the squares of all the integers in that structure." This led to *generic* programming (Sect. 3). Genericity was achieved by using *elementary* concepts from algebra and category theory — functors, initial and final algebras, and the various kinds of morphisms on them [xiv] (Sect. 4). Programs taking advantage of this are called "polytypic", i.e. allowing many kinds of type structures, in the same way that polymorphic programs allow many kinds of type values within a single class of structures.

Unfortunately, the kind of specification that most polytypic languages support in their type signatures is very limited. Type theory [xv] however showed how any specification expressible in predicate logic could serve as the *type* of a program. That enables programmers to capture arbitrary invariants and specifications of their programs, such as *balanced* trees or *sorted* lists, simply as part of the program's type. Since types are checked at compile-time, any type-correct program will never violate those specifications at runtime. This is supported by *dependently typed* programming languages (Sect. 5).

Besides the activities around data structures there was also a branch of work dealing with the task of mimicking imperative structures, as, e.g., necessary to describe interaction with the environment, in a purely functional context. *Monads*, *applicative functors*, and *algebraic effects* have provided a mathematically solid account that could be formulated in a way that allowed program-algebraic calculation after all (Sect. 6).

The investigations into data structures and generic algorithms on them were mainly carried out around (quotients of) tree-like structures. However, there are numerous more general (graph-like) structures which are not easily represented in that framework. As these should be approachable by calculations as well, our activities have therefore also dealt with relational or relationally based structures, which is their natural mathematical representation. Abstracting relations to algebraic structures such as Kleene algebras provides notions well suited for

```
*(y≠0 → z,x,y := z′,x′,y′ | z′,x′,y′:
    z•x^y = X^Y & y≠0 ⊃ z′•x′^y′=X^Y & y′<y).
```

```
*(y≠0 → z,x,y := z′,x′,y′ | z′,x′,y′,r:
    z′= z•x^r & x′= x•x & y=2y′+r &
        (r=0 ∨ r=1)).
```

Fig. 1. Abstracto 84 [xx]

describing not only data structures but also control structures of various kinds (Sect. 7). This approach also links nicely to the predicate transformer approaches [vi] and the "Laws of Programming" [viii].

Systematic program construction benefits greatly from program construction systems — tools to support the work of the program constructor. This work involves reasoning about programs, which can be shallow and tedious; automated tools are less error-prone than humans at such activities. Moreover, programs are usually much longer than formal expressions in other contexts, such as in traditional mathematics; so tool support is also a convenience. Finally, a system can record the development history, producing automatically the software documentation that allows a replay, upon a change of specification, or an audit if something goes wrong. The Group has always worked on design and engineering of transformation systems in parallel with the work on the underlying transformation calculi; our survey therefore concludes with a more detailed account of corresponding tool support (Sect. 8).

Generally, the Group's pattern has always been to expand the concepts that enable rigorous construction of correct programs, then streamline their application, and finally simplify their presentation. And then... expand again.

As the trajectory in this section has described (with the benefit of hindsight) the Group has always had diverse interests that arise from our program-calculational 'mindset' applied to other computer-science interest areas and even real-world contemporary problems [xvi].

2 From ALGOL, via Abstracto... to Squiggol

2.1 *Abstracto*: the first move towards algorithmics

After the completion of the *Revised Report on ALGOL 68* [xix], the Group set up a *Future Work* subcommittee to decide how to progress. This subcommittee in turn organised two public conferences on *New Directions in Algorithmic Languages* [xi], after which the Group focussed again on specific topics. The Chair highlighted two foci: programming languages for beginners [xvii], and "Abstracto". The first led to the development of the beginner's language ABC and hence eventually to Python [xviii]; the other was *Abstracto*, and was

> ... not a specification language as such since it is still concerned with how to do things and not just what is to be done, but [allowing] the expression of the 'how' in the simplest and most abstract possible way. [xi]

A representative example of Abstracto is shown in Fig. 1. It is part of the development of a 'fast exponentiation' algorithm: given natural numbers X and

```
input dm, mr;
gdb := ∅;
for m ∈ dm do
   gdb := gdb ∪ mr[m]
endfor;
aoi := −∞;
for i ∈ gdb do
   if i·age > aoi then
      oi, aoi := i, i·age
   endif
endfor;
output oi.
```

$\Longrightarrow$

```
input dm, mr;
slm := ∅;
for m ∈ dm do
   alm := −∞;
   for i ∈ mr[m] do
      if i·age > alm then
         lm, alm := i, i·age
      endif
   endfor;
   slm := slm ∪ {lm}
endfor;
aoi := −∞;
for i ∈ slm do
   if i·age > aoi then
      oi, aoi := i, i·age
   endif
endfor;
output oi.
```

Fig. 2. The oldest inhabitant, in Abstracto [136]

Y, compute $z = X^Y$ using only $O(\log_2 Y)$ iterations. The program on the left shows a 'while' loop, with invariant $z \times x^y = X^Y$, variant y, and guard $y \neq 0$. The program on the right factors out $r = y \bmod 2$, refining the nondeterminism in the first program to a deterministic loop. Thus our vision for Abstracto was as a kind of 'refinement calculus' for imperative programs [xxi].

2.2 The Bird–Meertens Formalism (BMF): A Higher-Level Approach

Although the Abstracto approach was successful, in the sense that it could be used to solve the various challenge problems that the Group worked on, after some time it was realised that the transformation steps needed were too low level — and so a key insight was to lift the reasoning to a higher level [xxii], namely to abandon the imperative ALGOL-like style and the corresponding refinement-oriented approach of Abstracto, and to switch instead to a more algebraic, functional presentation.

It made a big difference. Consider for example the two programs in Fig. 2 [xx], where the problem is to find the (assumed unique) oldest inhabitant of the Netherlands. The data is given by a collection dm of Dutch municipalities, and an array $mr[-]$ of municipal registers of individuals, one register per municipality. The program on the left combines all the municipal registers into one national register; the program on the right finds the oldest inhabitant of each municipality, and then findest the oldest among those 'local Methuselahs'. Provided that no municipality is uninhabited, the two programs have equivalent behaviour. However, one cannot reasonably expect that precise transformation, from the one to the other, to be present in any catalogue of transformations. Instead, the

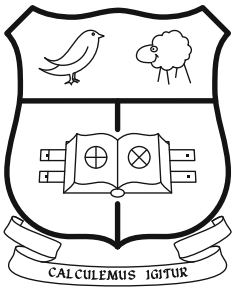
$$
\begin{array}{lcl}
mss & = & \text{definition} \\
 & & \uparrow/ \cdot +/* \cdot segs \\
 & = & \text{definition of } segs \\
 & & \uparrow/ \cdot +/* \cdot +\!\!+/ \cdot tails * \cdot inits \\
 & = & \text{map and reduce promotion} \\
 & & \uparrow/ \cdot (\uparrow/ \cdot +/* \cdot tails) * \cdot inits \\
 & = & \text{Horner's rule with } a \circledast b = (a + b) \uparrow 0 \\
 & & \uparrow/ \cdot \circledast /\!\!/_0 * \cdot inits \\
 & = & \text{accumulation lemma} \\
 & & \uparrow/ \cdot \circledast \#_0
\end{array}
$$

Fig. 3. The maximum segment sum problem [xxiv]

development should proceed by a series of *simpler* steps that, because of their simplicity, can feasibly be collected in a smaller and more manageable catalogue of general-purpose transformations.

The equivalent higher-level transformation is this one: [xxii]

$$\uparrow_{age}/ +/mr * dm \;=\; \uparrow_{age}/(\uparrow_{age}/mr) * dm$$



Let us calculate!

Its left-hand side takes the oldest in the union of the registers of each of the municipalities, and the right-hand side takes the oldest among those local Methuselahs. The "$\oplus/$" reduces a collection using binary operator $\oplus$; the "$+$" is binary union; the "$\uparrow_f$" chooses which of two arguments has the greater f-value; the "$g*$" maps function g over a collection; and finally, function composition is indicated by juxtaposition. The functional presentation is clearly an order of magnitude shorter than the Abstracto one. It is also easier to see what form the small general-purpose transformation steps should take—simple equations such as "reduce promotion" ($\oplus/ \; +/ \; = \; \oplus/ \; \oplus/*$) and "map fusion" ($f* \, g* \; = \; (f \, g)*$) [xxiii]. The notation evolved further through the 1980s [xxiv], and came to be known as "Squiggol". It was later given the more respectable name "Bird–Meertens Formalism" [xxv], and inspired the Group's further efforts in rigorous, concise program development.

Another example of concise calculation is given in Fig. 3.

2.3 The Boom Hierarchy of Data Structures

The operators and transformation rules of Squiggol/BMF apply equally to lists, bags, and sets. And those three datatypes are conceptually linked by their common signature of constructors (an empty structure, a singleton constructor, and a binary combination) but satisfying different laws (associativity, commutativity, and idempotence of the binary combination, with the empty structure as a

unit). Moreover, the core operations (maps, filters, and reductions) are homomorphisms over this algebraic structure.

Crucially, each datatype is the *free algebra* on that common signature, with a given set of equations, generated from a domain of individual elements; that is, there exists a *unique* homomorphism from the datatype to any other algebra of the same kind. For example, writing "[ ]" for the empty structure, "$[x]$" for a singleton, "$+\!\!+$" for the binary combination, and given a binary operator $\oplus$ with unit e, the three equations

$$\begin{aligned} \oplus/[\,] &= e \\ \oplus/[a] &= a \\ \oplus/(x +\!\!+ y) &= \oplus/x \oplus \oplus/y \end{aligned}$$

determine the reduction operator $\oplus/$ uniquely: provided that $\oplus$ is associative, these three equations have as their unique solution the aggregation function from lists. But if we add the assumption that $\oplus$ is also commutative, then there is a unique function from bags; and if we add idempotence, then there is a unique function from sets.

If out of curiosity we assert *no* equations of the binary operator alone, only that the empty structure is its unit, then we obtain a fourth member of the family, a peculiar sort of binary tree. The four members form a hierarchy, by adding the three equations one by one to this tree type. The resulting hierarchy of data structures was called the "Boom" hierarchy [xiii]. Its connections to the Eindhoven quantifier notation [xxvi] greatly simplified the body of operators and laws needed for a useful theory.

3 Generic Programming: Function Follows Form

The Boom hierarchy is an example of how we can use algebras and homomorphisms to describe a collection of datatypes, together with a number of basic operations on those datatypes. In the case of the Boom hierarchy, the constructors of the algebra are fixed, and the laws the operators satisfy vary. Another axis along which we can abstract is the *constructors* of a datatype: we realised that concepts from category theory can be used to describe a large collection of datatypes as initial algebras or final coalgebras of a functor [xiv]. The action of the initial algebra represents the constructors of the datatype it models. And it has the attractive property that any homomorphism on the functor algebra induces a unique function from the initial algebra. Such a function was called a *catamorphism* [xxvii]. A catamorphism captures the canonical recursive form on a datatype represented by an initial algebra. In the functional programming world, a catamorphism is called a fold, and in object-oriented programming languages the concept corresponds closely to visitors. Typical examples are functions like map, applying an argument function to all elements in a value of a datatype, and size, returning the number of elements in a value of a (container) datatype. Catamorphisms satisfy a nice fusion property, which is the basis of many laws in programming calculi. This work started a line of research

```
flatten  :: Regular d => d a -> [a]
flatten  =  cata fl

polytypic fl :: f a [a] -> [a] =
  case f of
    g + h -> either fl fl
    g * h -> \(x,y) -> fl x ++ fl y
    ()    -> \x -> []
    Par   -> \x -> [x]
    Rec   -> \x -> x
    d @ g -> concat . flatten . pmap fl
    Con t -> \x -> []

data Either a b  =  Left a | Right b
```

Fig. 4. A PolyP program to flatten a container to a list [xxix]

on *datatype-generic programming* [xxviii], capturing various forms of recursion as morphisms, more about which in Sect. 4.

The program calculus thus developed could be used to calculate solutions to many software problems. As a spin-off, the theory described programs that could be implemented in a standard, but different, way on datatypes that can be described as initial functor-algebras. No general-purpose programming language supported such typed, generic functions, so these functions had to be implemented over and again for different datatypes.

Using the structure of polynomial functors, the language PolyP was designed that extended the lazy, higher-order functional programming language Haskell [xxix]. In PolyP, a generic function is defined by means of induction on the structure of functors. Using this programming language it was possible to define not only the recursive combinators from the program calculus, such as folds and unfolds, but also to write generic programs for unification, term rewriting, pattern matching, etc. Figure 4 shows an example of a polytypic program.

PolyP supported the definition of generic functions on datatypes that can be described as initial functor-algebras but do not involve mutual recursion. While sufficient for proof-of-concept demonstration purposes, this last restriction was a severe limitation on practical applicability. Generic programming is particularly attractive in situations with large datatypes, such as the abstract syntax of programming languages, and such datatypes are usually mutually recursive. Generic Haskell was developed to support generic functions on sets of mutually recursive datatypes [xxx]. Generic functions defined in Generic Haskell can be applied to values of almost any datatype definable in Haskell. Figure 5 shows how a generic equality function is implemented in Generic Haskell.

The approach of defining generic functions in Generic Haskell can also be used to define type-indexed (or generic) datatypes. A type-indexed datatype is a data type that is constructed in a generic way from an argument data type. For example, in the case of digital searching, we have to define a search tree type

$$
\begin{array}{ll}
\textbf{type}\ \mathsf{Eq}\{\![\star]\!\}\ \mathsf{t} & = \mathsf{t} \to \mathsf{t} \to \mathsf{Bool} \\
\textbf{type}\ \mathsf{Eq}\{\![\kappa \to \nu]\!\}\ \mathsf{t} & = \forall \mathsf{a}\,.\,\mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{a} \to \mathsf{Eq}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a}) \\
\\
eq\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{t} \\
eq\{\!|\mathsf{Char}|\!\} & = eqChar \\
eq\{\!|\mathsf{Int}|\!\} & = eqInt \\
eq\{\!|\mathsf{Unit}|\!\}\ Unit\ Unit & = True \\
eq\{\!|:+:|\!\}\ eqa\ eqb\ (Inl\ a)\ (Inl\ a') & = eqa\ a\ a' \\
eq\{\!|:+:|\!\}\ eqa\ eqb\ (Inl\ a)\ (Inr\ b') & = False \\
eq\{\!|:+:|\!\}\ eqa\ eqb\ (Inr\ b)\ (Inl\ a') & = False \\
eq\{\!|:+:|\!\}\ eqa\ eqb\ (Inr\ b)\ (Inr\ b') & = eqb\ b\ b' \\
eq\{\!|:*:|\!\}\ eqa\ eqb\ (a :*: b)\ (a' :*: b') & = eqa\ a\ a' \wedge eqb\ b\ b'
\end{array}
$$

Fig. 5. A generic Haskell program for equality [xxx]

by induction on the structure of the type of search keys. Generic Haskell also supports the possibility of defining type-indexed datatypes [xxxi]. The functional programming language Haskell now supports a light-weight variant of type-indexed datatypes through type families.

The fixed-point structure of datatypes is lost in Generic Haskell, however, and with it the capability of defining the generic fold function. It was then discovered how to obtain a fixed-point representation of possibly mutually recursive datatypes, bringing the generic fold function back into the fold [xxxii]. Thus we can define the fold function for the abstract syntax of a programming language, bringing generic programming within reach of compiler writers.

Meanwhile, Haskell –or, more precisely, compilers supporting various Haskell extensions– evolved considerably since PolyP and Generic Haskell were developed. With respect to types, GHC, the Glasgow Haskell Compiler, now supports multiple-parameter type classes, generalised algebraic datatypes (GADTs), type families, etc. Using these extensions, it is now possible to define generic functions in Haskell itself, using a library for generic programming. Since 2000, tens of such libraries have been developed world-wide [xxxiii]. Since –from a generic programming perspective– the expressiveness of these libraries is almost the same as the special-purpose language extensions, and since such libraries are much easier to develop, maintain, and ship, these libraries make generic programming more generally available. Indeed, these libraries have found their way to a wider audience: for example, Scrap Your Boilerplate has been downloaded almost 300,000 times, and Generic Deriving almost 100,000 times [xxxiii].

4 Morphisms: Suddenly They Are Everywhere

In Sect. 3 we identified catamorphisms as a canonical recursive form on a datatype represented by an initial algebra: in functional-programming parlance, a *fold*. From there, however, further work [xxxiv] led to a rich research agenda concerned with capturing the pattern of many other useful recursive functions

that did not quite fit that scheme, that were not *quite* 'catamorphic'. Indeed, it gave rise to a whole zoo of morphisms: *mutu*morphisms, *zygo*morphisms, *histo*morphisms, generalised folds, and generic accumulations [xxxv]. Just as with catamorphisms, those recursion schemes attracted attention because they made termination or progress manifest (no need to prove or check it) and they enjoyed many useful and general calculational properties — which would otherwise have to be established afresh for each new application.

4.1 Diversification

Where while-loops are governed by a predicate on the current state, and for loops by an incrementing counter, structured recursion schemes such as catamorphisms take a more restricted approach where it is the structure of the input data itself that controls the flow of execution ("function follows form").

As a simple example, consider how a list of integers is summed: a catamorphism simply recurses over the structure of the list. No for-loop index variable, and no predicate: when the list is empty the sum is zero, and when the list contains at least one number it should be added to the sum of the residual list. While-loops could easily encode such tasks, but their extra expressive power is also their weakness: we know that it is not always tractable to analyse loops in general. With catamorphisms, the analysis is much simpler — the recursion scheme is simply induction over a datatype.

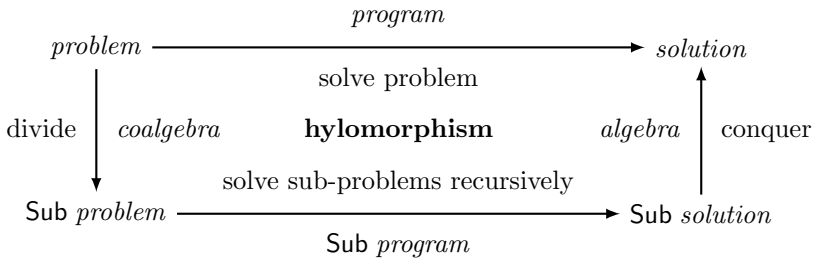
The analogy with induction goes further. Number theorists have long studied computable functions on natural numbers, and an important class are the primitive recursive functions, which provide the recursive step with the original argument as well as the result of recursing on that argument. Such functions are an instance of the *paramorphism* [xxxvi], which is an interchangeable variation on the catamorphism.

Further still, an attractive variant of induction is *strong* induction, where the inductive step can rely on all the previous steps. Its parallel as a recursion scheme is the *histomorphism* and, just as strong induction and induction are interchangeable, histomorphisms are encodable as catamorphisms. The utility of these schemes –the point of it all– is however to make it convenient to describe programs that would otherwise be difficult to express, and to derive others from them. In the case of histomorphisms (strong recursion), for example, it is the essence of simple dynamic programming programs such as the knapsack problem, or counting the number of possible bracketings, that was captured. More complex dynamic programming problems, such as the multiplication of a chain of matrices, requires a slightly more nuanced recursion scheme, the *dynamorphism*, where an intermediate data structure is generated.

We recall that the exploitation of various forms of duality revolutionalised the field of physics; algorithmics similarly benefits from an important form of input-output duality. Each recursion scheme features a dual scheme: while one focuses on consuming the input, the other emphasizes producing the output. To illustrate, consider how insertion sort deconstructs a list by extracting numbers one at a time (input), inserting them appropriately into a sorted list (output).

Whereas the deconstruction of the original list is another catamorphism, the construction of the sorted list exemplifies an *anamorphism*—it is the dual situation. Thus expressing insertion sort in terms of recursion schemes allows us to dualize the algorithm to obtain another sorting algorithm *for free*: selection sort. This works by constructing a sorted list (an anamorphism), and at each step performs a selection that deconstructs the unsorted list to extract the smallest element (a paramorphism).

Another way to understand a catamorphism is that it applies a strategy that takes subsolutions and conquers them (with a so-called *algebra*) to provide a final solution. Dually, an anamorphism applies a strategy that takes a problem and splits it up into subproblems (with a so-called *coalgebra*). Those can be understood as the two components of a divide-and-conquer strategy, and the combination is known as a *hylomorphism*, depicted in the diagram below:



Catamorphisms are then the special case of this diagram where the dividing step simply *deconstructs* a data structure, and anamorphisms the special case where the conquering step *constructs* a data structure.

4.2 Unification

The multitude of generalisations of catamorphisms and their duals is bewildering.

Many of them were defined as adaptations of catamorphisms, but in most cases showing that those corresponded directly to catamorphisms required careful calculation. And with so many different variations, a natural question is whether there is some underlying commonality that unifies them all. Indeed there is.

The unification was achieved by borrowing some slightly more sophisticated machinery from category theory. A first attempt was to use comonads, which allow access to contextual information [xxxvii], to organise the structure of recursive calls. Another attempt used adjunctions instead as the common thread [xxxviii]. That resulted in so-called "adjoint" folds, which show how a catamorphism in one category can give rise to a different recursion scheme in another. Although the two methods were initially thought to be disjoint, later work revealed recursion schemes from comonads to be a special case of adjoint folds with an appropriate distributive law.

Each of these two unifications of recursion schemes treated generalizations of catamorphisms separately to their dual counterparts of anamorphisms. But both

are special cases of hylomorphisms; and so the next step was to generalise *all* inductive and coinductive recursion schemes within the single unifying theme of *conjugate hylomorphisms* — or 'the mother of all recursion schemes'. Naturally, the Group named it the *mamamorphism*. This time, the more sophisticated categorical techniques were used to extend the work on adjoint folds with conjugate distributive laws to connect pairs of adjunctions.

All in all, the unifying work on recursion schemes benefitted greatly from the unifying power of category theory — which is what category theory is for.

5 Dependent Types: Types You Can Depend on

Datatype-generic programming explores how to define functions and datatypes by induction over the structure of algebraic types. This line of research within the Group sparked further interest in the exploration of how to use static type information in the construction of programs. In particular, emerging programming languages with *dependent types* offered new opportunities for program verification, program transformation, program calculation and type-directed program development.

5.1 What Are Dependent Types?

The idea of programming with dependent types dates back at least as far as the 1970's, when it became increasingly clear that there was a deep connection between constructive mathematics and computer programming [xxxix]. In the late 20th century, a number of new programming languages emerged, exploring these ideas [xl]. Those languages, and their implementations, enabled the further exploration of the possibilities that statically typed languages with dependent types offered. Each of them adopted the *Curry-Howard correspondence* [xli], connecting programming languages and mathematical logic, as the guiding principle of *program language design*. The terms of each language correspond to both programs and proofs; a type can equally well be read as a specification or a proposition. To ensure the logic underlying a language's type system is *sound*, all functions must be total, disallowing partial incomplete pattern matching and diverging functions. The benefit of this disciplined approach to software development is that these languages provide a unified setting for both programming and program verification. Given the strong traditions of program calculation and functional programming within the Group, for instance, using the Bird–Meertens Formalism to perform equational reasoning about Haskell programs, there was a clear interest in these new languages. Furthermore, the richer language of algebraic data types offered the ability to enforce invariants during a program's construction.

5.2 Dependent Types

At the beginning of the 21st century, the relation between dependently typed programming and datatype generic programming was clearly emerging [xlii] leading

to several influential PhD theses on this topic. The interest in dependent types from members of the Group dates back to the late 80's [xliii].

The new languages based on type theory reinvigorated some of the past research that members of the Group have done on the derivation of correct programs. Following the Agda tutorial at Meeting #63 [xliv], the work on relational program calculation, for example, was shown to be possible within dependently typed languages. Similarly, the refinement calculus, used to derive a program from its specification, could be embedded in a proof assistant, enabling pen and paper proofs to be machine-checked. Program calculation in the style of Dijkstra using predicate transformer semantics could be modelled using type theory, rather than the traditional impredicative set theory. Types and proof assistants based on type theory became a central tool in the calculation of correct programs [xlv].

At that point, an influx of new members broadened the Group's interest to novel application areas for dependently typed programming [xlvi], such as scientific computation, decision problems, and even the study of climate change. Combinator parsing, previously studied in the context of functional programming (see Subsect. 6.2), was implemented in a total language with dependent types [xlvii].

The new languages with dependent types also enabled new opportunities to exploit static type information to guide program development [xlviii] — in the same spirit as the research on datatype generic programming. Types can be read as a (partial) specification. The discovery of a type-correct program can arise from a dialogue with the type checker, helping establish a program's correctness as it is written. There are numerous domain-specific languages and data types designed to enforce certain correctness properties by construction.

Dependently typed programming languages marry constructive logic and programming in a manner unfamiliar to most programmers. To ensure that the type system is sound, all programs must be total. Yet any mainstream language relies on numerous *effects*, such as general recursion, mutable state, concurrency, or exceptions, each of which break the promise of totality. To address this, there has been a line of research on how to incorporate effects in dependently typed program languages [xlix]. This, in turn, led to renewed interest from the Group on how to program safely and robustly in the presence of arbitary side-effects in any language, resulting in the study of *algebraic effects* (see Sect. 6).

6 Computational Effects: Beyond the Functional

When the Group switched to a purely functional presentation of programs [xxii], that is from Abstracto to Squiggol (Sect. 2), at first this also meant doing away with a group of programming-language features known collectively as "effects".

6.1 Effects and Monads

Effects cover all behavioural aspects of a computational function that go beyond the input-output behaviour of mathematical functions. It includes interaction

of a program with its environment (the file system and operating system, other processes, human operators, distant servers, ...), mechanisms for structuring the internal control flow (partiality and exceptions, backtracking, nondeterminism and probability, delimited control, ...), and implicit dataflows (mutable state and global variables).

While some of these effects are indeed symptoms of a low-level imperative encoding, such as local mutable state, others are essential in real-world programs that interact with the environment. And they can be important for structuring programs compositionally: examples are exceptions and backtracking.

Fortunately, it turned out that useful effects need not be abandoned in a purely functional setting [l]—the 'doing away with' was only temporary. Effects can after all be modelled with pure functions. Here are some examples:

$a \to b$	a pure function
$a \to 1 + b$	a partial function
$a \to e + b$	a function with exceptions e
$a \to b^+$	a nondeterministic function
$a \to b^*$	... which might also fail
$a \to b \times o^*$	a function that sends o's to its environment
$a \to \mu x.((i \to x) + b)$	a function that reads i's from its environment
$a \to (s \to (b \times s))$	a function with implicit state s
$\vdots$	

(where b^+ denotes non-empty sequences of b's, and b^* possibly empty sequences).

It turned out that all those different types of functions with effects are 'Kleisli' arrows for appropriately structured *monads* [li]. The utility of the monad was that it handled calculation, in particular composition, of the types above in a single unified way. Whereas two functions of types $a \to b$ and $b \to c$ are easily composed to make a single function of type $a \to c$, it is not clear at first how to compose $a \to e+b$ and $b \to e+c$ to $a \to e+c$, or for that matter $a \to b^+$ and $b \to c^+$ to $a \to c^+$. And even when the (in retrospect) obvious definitions are adopted, one for each, the challenge is then to see those definitions as instances of a single generalised composition. That's what Kleisli composition achieves.

6.2 Functions Too Weak, Monads Too Strong: Applicative Functors? Just Right

Once monads had brought effects back in the purview of purely functional reasoning, the Group turned its attention to reasoning about such programs—'effectful' programs. One fruitful example has been the study of *recursive descent parsers* [lii]. They lend themselves to a combinator style of programming. Moreover, the combinators fall neatly out of the observation that the datatype of parsers that return a parsed value is another monad, a combination of implicit state and nondeterminism with failure: the Kleisli arrows are of the form

$$a \to (\Sigma^* \to (b \times \Sigma^*)^*)$$

where the alphabet of symbols is Σ or, in verse [liii],

A parser for things
is a function from strings
to lists of pairs
of things and strings.

But the monadic presentation makes static analysis difficult: the interface allows earlier inputs to determine the parser used for later inputs, which is both more expressive than necessary (because few applications require such configurable syntax) and too expressive to analyse (because the later parser is not statically available). A weaker interface for effects turns out to be nearly as expressive, and much more amenable to analysis. The essence of this weaker interface was abstracted as an 'applicative functor', and has served as the foundation of significant subsequent work [liv].

6.3 Algebraic Effects and Handlers

But how to reason about effectful programs, such as applicative parsers, nondeterministic functions, and programs that perform I/O? A first step is to treat the effectful operations as an abstract datatype, provide a purely functional specification of that data abstraction, prove the program correct with respect to the algebraic specification, but run the program against the 'real' implementation that incurs actual effects such as I/O. In fact, one could consider the algebraic specification as the interface in the first place, and incorporate its axioms into traditional equational reasoning; it is then the responsibility of the implementer of the effect to satisfy the axioms. This approach is cleanly formalized in the notion of *algebraic effects and handlers*, whereby a pure functional program assembles a term describing the effects to be performed, and a complementary environment *handles* the term, by analogy with handling an exception [lv]. In fact, that term is a value of a type captured as the *free monad* on the signature of the effect operations, a datatype-generic notion (see Sect. 3).

7 Lifting the Game: A Purely Algebraic View of Algorithms and Languages

The systematic construction of algorithms –or, more generally, of computer programs– needs languages that are precise, effective, and that allow calculational reasoning. Previous sections showed how the Group discovered the striking similarities between derivations from quite different areas, such as path problems and regular languages [lvi]. Using algebra in its purest form, i.e. starting with a collection of postulated axioms and carrying out (program) derivations based on those laws alone, therefore enables an extremely abstract treatment: those derivations are then valid in *any* programming model that satisfies the axioms.

Calculi based on the algebra of binary relations [lvii] were prime candidates for that, since they allow a natural treatment of directed graphs—and they abstract and unify data structures (e.g. trees), transition systems and many more concepts.

	semiring (program interpretation)		relation algebra
$+$	(nondeterministic) choice	$\cup$	union
$\cdot$	sequential composition	$;$	relational composition
$\leq$	refinement	$\subseteq$	subset
0	abort	$\emptyset$	empty relation
1	skip	I	identity relation

Fig. 6. Operators of semirings and relation algebras

Also, relations are intimately connected with predicates and hence can be used to describe (by pre- and postconditions) and calculate input-output behaviour. In particular, they cover principles of algorithm design such as dynamic programming, greedy algorithms etc. [lvi]

Relation Algebras make relations, i.e. sets of argument-value pairs, 'first-class citizens' by viewing them as algebraic elements subject to operators that treat them as a whole without looking at their internal structure. The 'point-free' approach that this enables often admits considerable concision. The basic relational operators (Fig. 6, right) are simply set union, intersection and complement, supplemented by sequential composition.

Although a relation-algebraic approach already allows the unified treatment of different instances of graph problems [lviii], replacing sets of pairs (single relations) by other entities yields further models of the same algebraic signature, known as *(idempotent) semirings*. Figure 6 (left) shows the operators common to semirings.

And those structures have applications in programming languages, algorithms, logic and software engineering:

- *Classical logic* is a well known simple semiring, in which choice corresponds to disjunction, composition to conjunction, 0 to false and 1 to true. To subsume classical logic fully, however, one requires negation — i.e. a Boolean algebra.
- When elements of a semiring are interpreted as (arbitrary) *programs*, the basic operators represent nondetermistic choice and sequential composition; 0 corresponds to the program abort and 1 to skip. Equations such as $1 \cdot x = x = x \cdot 1$ and $0 \cdot x = 0 = x \cdot 0$ form the basis of algebraic reasoning, including program transformations. The equations describe the facts that any program x composed with skip is identical to the program itself, and that any program composed with abort is identical to abort. This allows the expression of programs and specifications in the same framework. A program P satisfies a specification S if $P \leq S$, where $\leq$ expresses refinement, which is the canonical order available in every idempotent semiring. (In other styles of program calculation, that would be written $S \sqsubseteq P$.) This simple formulation of program correctness enables a wide range of methods for calculational program derivation and program verification [lix].
- Using partial maps as algebraic elements allows treating data structures with pointers. This usage was inspired by Squiggol (Sect. 2) [lx].

- When the underlying structure reflects the memory cells (heaps), the algebraic framework provides an abstract version of separation logic [lxi].
- When the algebraic elements are interpreted as sets of sets or sets of lists it is possible to derive aspects of feature-oriented software development, including the formal characterisation of product families and of feature interactions [lxii].
- Graphs are often equipped with edge labels representing weights, capacities or probabilities; likewise automata and labelled transition systems carry extra edge information in addition to source and target. Those can be treated by generalising Boolean matrices to matrices over other algebras. For classical graph algorithms, such as shortest-path algorithms, the max-plus algebra and the min-plus algebra are useful as underlying structure—here, min/max play the roles of (biased) choice, and plus is the operator for sequential composition (that is, adding path lengths/costs).
- Probabilistic aspects can be represented by matrices with real values between 0 and 1, and fit into the very same algebraic framework. Applications include calculational derivations of fuzzy algorithms.
- Fundamental concepts of programming-language semantics, including concurrent programs and termination, can be handled algebraically as well. Beyond the areas mentioned above, the Group has also applied this algebra in several areas, included object-oriented programming, data processing, game analysis and routing in mobile networks [lxii].

But semirings can be extended: and those extensions are used to capture additional concepts from data structures, program logics and program transformation. Here are some examples.

Kleene algebras, generalising the algebra of regular expressions, offer the additional operator $_^*$ of arbitrary finite iteration. Algebraically, the loop $\mathsf{while}\ p\ \mathsf{do}\ x$ becomes $(p \cdot x)^* \cdot \neg p$, which is the least fixed-point of the function $\lambda y.\ \mathsf{if}\ p\ \mathsf{then}\ x \cdot y\ \mathsf{else\ skip}$ [lxiii].

Here p is a specific element, called a *test*, representing a predicate on the state space. The set of tests offers a negation operator $\neg$ and hence forms a Boolean algebra [lxiv]. In the interpretation where algebraic elements are programs, a test p corresponds to an assert statement. For tests p, q and program element x the inequation $p \cdot x \leq x \cdot q$ algebraically expresses the Hoare triple $\{p\}x\{q\}$ [lxi].

Furthermore, in certain Kleene algebras, known as *quantales*, the principle of fixed-point fusion [lxv] is a theorem, i.e. it can be derived from the axioms. This illustrates once again the powers of 'algebraic unification'. Fusion, shown in Sects. 2 and 3 to be an extremely practical law for transforming functional programs, is now available for many other kinds of program too. Examples include merging of programs with the same loop structure, or 'deforestation', i.e. avoiding the generation of a large intermediate data structure that afterwards is 'consumed' again, in favour of 'generation and consumption on the fly'. This is also known as "virtual" data structures [lxvi].

Omega algebras [lxvii], which offer an operator $_^\omega$ for infinite iteration, allow the description and analysis of systems or programs with potentially never-ending behaviour, such as operating systems.

In algebras with finite and infinite behaviour, some algebraic laws of sequential composition need to be adapted by separating the finite and the infinite traces of a program x into the disjoint sets $\mathsf{fin}\, x$ and $\mathsf{inf}\, x$. While the above law $x \cdot 1 = x$ still holds for all elements, the property $x \cdot 0 = 0$ no longer holds when x contains infinite traces; it weakens to $(\mathsf{fin}\, x) \cdot 0 = 0$. The intuitive explanation is that infinite traces do not terminate, and therefore a possible successor, including abort, can never be 'executed'. Therefore the while-loop now has the more general behaviour

$$(p \cdot x)^* \cdot \neg p \quad = \quad (p \cdot \mathsf{fin}\, x)^* \cdot (\neg p + p \cdot \mathsf{inf}\, x) \quad ,$$

which means that after a finitely many finite traces from x the loop either terminates by not satisfying the test p any longer, or an infinite trace from x takes over, leading to overall non-termination. When x is purely finite, i.e., satisfies $\mathsf{inf}\, x = 0$, this reduces to the expression given previously.

Like the operators of semirings, the operators of finite and infinite iterations (and many of their combinations) satisfy a common set of laws, and thus algebra helps to unify their treatment including the derivation of program transformations and refinement theorems. Applications range from termination in classical programs, via protocols, to dynamic and hybrid systems [lxvii].

Omega algebras are also used to develop a unified framework for various logics, including the temporal logics LTL, CTL and CTL*, neighbourhood logic and separation logic [lxi].

To sum up: algebraic characterisations have helped to express (and prove) new notions and results and to unify concepts and identify the above-mentioned similarities. The Group has developed a coherent view on algorithms and languages from an algebraic perspective, and applies the same algebraic techniques to tackle modern technology, including the analysis of protocols and quantum computing. All the algebras in question provide a first-order equational calculus, which makes them ideal to be supported by *automated theorem provers* and *interactive proof assistants* [lxviii] [xliv]. As a consequence, they are well suited for developing tools that support program derivations and refinement in a (semi-)automated style.

8 System Support: the Right Tool for the Job

Calculational program construction derives a program from a formal specification by manageable, controlled steps that –*because* they are calculated– guarantee that the final product meets its initial specification. As we have seen, this methodology has been practised by many Group members, and many others too [lxix]. And it applies to many programming styles, including both functional and imperative. For the former one uses mostly equational reasoning, applying the defining equations of functions together with laws of the underlying data

structures. For the latter, inequations deploying a refinement relation are common [lxx]. A frequent synonym for "calculation rules" is "transformation rules".

A breakthrough occurred when the Group raised the level of reasoning (Sect. 2): from manipulations of imperative code (Abstracto) to algebraic abstractions of functional control patterns (Squiggol). This made it possible to compact derivations of several pages in traditional approaches down to one page or even less. A similar observation concerns the general theme of 'algebraicisation' (see Sect. 7).

8.1 System Support

Of course, calculational program construction can be done with pencil and paper, and initially it should be so: that encourages a simplicity and elegance in its methods. Ultimately, if the method proves to be useful, there are a number of good reasons for introducing system support:

- By its very nature, program transformation leads to frequent rewritings of program fragments; such clerical work should be automatic. And, by *its* very nature, a system does this mechanical activity better than a human can.
- The system can record the applicability conditions and help in reducing them to simpler forms, ideally all the way to "true".
- And, as mentioned in Sect. 1, the system can construct a *development history*, again a clerical task. This history serves as detailed software documentation, since it reflects every design decision that enters into the final program. Thus, if a line of development turns out to be a blind alley, the history can be used for backtracking to try out alternative design decisions. Moreover, it is the key aid to software maintenance: when the specification has to be modified (because of new requirements), one can try to 'replay' a recorded development accordingly.

Thus the Group gave considerable attention to program transformation systems [ix] once the methods they automated were sufficiently mature. In the remainder of this section we take a brief look at one of them: it touches on several areas within the Group, and several Group members were involved in it and in follow-on projects.

8.2 An Example: The Project CIP

The project CIP (*Computer-aided, Intuition-guided Programming*) at TU Munich ran roughly through the period 1977–1990.

The Wide-Spectrum Language CIP-L. The CIP approach was based on a particular 'life cycle of transformational program development', roughly characterised by the following levels [lxxi]:

1. formal problem specification (usually descriptive, not (yet) executable, possibly non-deterministic);

2. recursive functional program;
3. efficiency-improved functional program;
4. deterministic, tail-recursive solution;
5. efficient procedural or machine-oriented program.

However, not all of these levels need occur: a development may start below Level 1 and end above Level 5; and it may skip some of the intermediate levels.

The language CIP-L was however especially designed to cover all five levels [lxxii]. Since transformations usually do not change a program as a whole, only small portions of it, it was mandatory to design one integrated wide-spectrum language rather separate languages for each level. In particular, the language included assertion constructs at all levels, thus allowing the incorporation of pre- and postconditions uniformly for functions and statements — so it is also connected to the refinement calculi that were developed around the same time [lxx]. CIP-L was partly inspired by Abstracto (Subsect. 2.1); in a sense, it tried to present a model of a possible concrete instance of Abstracto.

The Transformation System CIP-S. The purpose of CIP-S was the transformational development of programs and program schemes. In addition to bookkeeping tasks, that included the manipulation of concrete programs, the derivation of new transformation rules within the system, and support for the verification of side conditions of transformation rules [lxxiii].

In keeping with the overall CIP methodology, the kernel of the system was itself formally specified: starting from that specification, all routines were developed to Pascal-level CIP-L using an earlier prototype system. The results were embedded into an appropriate user environment, yielding a first operational version of CIP-S around 1990. In conjunction with a compiler for a substantial executable subset of CIP-L, the CIP-S system has been successfully used in education. The transformational approach was continued by the Group.

Experiences. There is an extensive body of case studies using the CIP methodology. They concern mostly small and medium-sized algorithms, e.g., sorting and parsing [lxxiv]. The formal development of CIP-S itself showed that the method is suitable for larger software projects too.

9 Summary; but No Conclusion

This is not a 'conclusion'. And this article is not a history. It is a description of a *goal*, a justification of its importance, and a summary of the trajectory that has led, and still leads to progress towards that goal. And what we especially enjoy about that trajectory we have followed, right from the start 60 years ago, is that it has always been the same one:

Let us calculate! (Sect. 2 p6)

Why is that goal so important?

Writing programs using a careful process of walk-throughs and reviews is (alone) not enough; "growing" programs [lxxv] in a top-down way is (alone) not enough; proving your program correct afterwards is (alone) not enough. We have always believed that maintaining correctness from the very top, and then 'all the way down' is what we all should be aiming for.

But will we ever get there? *No, we will not.*

During the 1970's, an array-out-of-bounds error in a high-level program would typically lead to a core dump, an inch-high stack of paper that was examined at just one spot, an "Ah, yes!" and then the whole thing just thrown away. Thirty years of progress brought us to 'Interactive Development Environments' and the internet, where sometimes the programmer was not even sure *where* the just-corrected version of a program had been 'deployed', nor exactly *whether* it contained the fix (because of caching). Error messages from a remote server in some far-away city flicked up out of the window, too quickly to be read, and could not be scrolled back. And twenty more years bring us up-to-date, with 'intelligent' aquarium thermometers that can be hacked from half a world away and used to raid a company's private database. *Plus ça change...*

The one constant through all of this is *people*, their tolerance for impediments to getting their work done and their perseverance in spite of them. The technology we are trying to control, to approach rigorously, is always sitting on that boundary, just beyond our reach: we will never calculate far enough.

Thus, however good we become at calculating, and convincing others to do so, there will always be economic forces that promote and propagate computer applications that we *cannot* develop by careful walk-throughs, or grow top-down, or prove correct... or calculate. This 'catching up' factor is what drives all the IFIP working groups — we constantly extend our methods improve the impact of computers generally, to make them safer and increase their reliability, as their use becomes ever more ambitious and spreads ever more widely.

We are not so much 'pushing' as 'being pulled'. There is the excitement.

10 Detailed Attributions and Citations

[i] **Contributors** —
All of the members members of WG2.1, past and present, deserve credit for what is reported here. Among those who provided actual text were Richard Bird, Jeremy Gibbons, Ralf Hinze, Peter Höfner, Johan Jeuring, Lambert Meertens, Bernhard Möller, Carroll Morgan, Tom Schrijvers, Wouter Swierstra and Nicolas Wu.
Carroll Morgan was Group Chair at the time of writing, and is the corresponding author.

[ii] **The founding of IFIP** —
It was established on 23 March 1962 [26,158].

[iii] **Change of name** —
At Meeting #39 in Chamrousse in January 1989, Formal Resolution 2 was to recommend to TC2 that the Group's name be changed to "WG2.1 on

ALGOL: Algorithmic Languages and Calculi". But TC2 rejected the recommendation, as reported at Meeting #40. At Meeting #41 in Burton in May 1990, it was reported that TC2 suggested instead simply "Algorithmic Languages and Calculi", and this suggestion was accepted by the Group. TC2 approved the change, which was reported at Meeting #42 in Louvain-la-Neuve in January 1991.

[iv] **Assigning meanings to programs** —
This was Floyd's association of predicates with flowchart arcs [70].

[v] **An axiomatic basis for computer programming** —
This was Hoare's logic for partial correctness [95].

[vi] **A Discipline of Programming** —
This was Dijkstra's calculus of weakest preconditions [65].

[vii] **Predicative programming** —
This generalisation was the work of Hoare and Hehner [87,88,96].

[viii] **Laws of Programming** —
This work was presented by a number of authors, including Hoare, at Oxford's Programming Research Group [97].

[ix] **Program-transformation systems** —
Systems designed and implemented by Group members include the Argonne TAMPR (Transformation-Assisted Multiple Program Realization) System [41–43], ARIES (Acquisition of Requirements and Incremental Evolution of Specifications) [113], (R)APTS (Rutgers Abstract Program Transformation System) [162], KIDS (Kestrel Interactive Development System) [185], POPART (Producer of Parsers And Related Tools) [201,202], ZAP [67,68], and the Munich CIP (Computer-aided, Intuition-guided Programming) project [21,23,149]. Comparisons of various transformation systems are presented in [69,170].

[x] **The name "Abstracto"** —
The lecturer who made that remark was Leo Geurts [73, p57]; he added that "in abstracto" was Dutch [sic!] for "in the abstract".

[xi] **Criteria for Abstracto** —
These criteria for Abstracto were proposed by Robert Dewar, who was the Group's chairman at the time [64]. His letter was written in July 1977 [64], in advance of Meeting #23 of the Group in Oxford in December of that year. The *New Directions in Algorithmic Languages* conferences were in 1975 and 1976, the work of a subcommittee chaired by Robert Dewar and with proceedings [181,182] edited by Stephen Schuman.

[xii] **Abstracting Abstracto** —
This landmark step was suggested and developed by Richard Bird and Lambert Meertens.

[xiii] **The Boom Hierarchy** —
The Boom hierarchy was introduced by Hendrik Boom [38], and thus namesd "Boom" (by others) — another pun, since Hendrik is Dutch, and "boom" is Dutch for tree. Backhouse [11] presents a detailed study of the Boom Hierarchy, and compares it to the quantifier notation introduced by Edsger Dijkstra and colleagues at Eindhoven.

[xiv] **The appeal to category theory** —
The introduction of concepts from category theory was due to Grant Malcolm [126], based on the work of Hagino [86].

[xv] **The connection between type structure and data structure** —
This observation was made by Martin Löf [130], and later by many others, including by Roland Backhouse in his work on type theory [13].

[xvi] **The Group's diverse interests** —
Our methods have been applied to separation logic [56], pointer structures [34,142], database queries [79,146], geographic information systems [145], climate change [39,108,110], scientific computation [109], planning [36] and logistics [172], and domain-specific languages for parsing/pretty printing/program calculation.

[xvii] **Beginner's programming languages** —
Beginner's programming languages designed and implemented by Group members include Peter King's *MABEL*, Kees Koster's *ELAN*, and Lambert Meertens' *ABC* [74].

[xviii] **Inspiration for Python** —
ABC's influence on Python [176] can be seen at Guido van Rossum's biographical page, and at the ABC and Python pages on Wikipedia:
https://gvanrossum.github.io/bio.html
https://en.wikipedia.org/wiki/ABC_(programming_language)
https://en.wikipedia.org/wiki/Python_(programming_language)

[xix] **Revised Report on ALGOL 68** —
ALGOL 68 was designed by WG2.1 at the direction of TC2. On December 20, 1968, the language was formally adopted by the Group, and subsequently approved for publication by the General Assembly of IFIP.

[xx] **Example of Abstracto** —
This example is from Lambert Meertens [135].

[xxi] **Refinement calculus** —
The 'Abstracto vision' was Lambert Meertens'. It was developed in much greater depth by Ralph Back (independently) [9,10] and, later, by Carroll Morgan [151,152]. When Morgan asked Meertens why he had not pursued the refinement calculus further, Meertens' reply was "It didn't work."

[xxii] **Higher-level reasoning** —
Meertens became disillusioned with Abstracto's low-level transformations, as described in [137]. It was Richard Bird who provided the key insight needed to lift the reasoning to a higher level [30]. Examples are given in [136].

[xxiii] **Program transformations** —
These examples, and many others, were described by Bird [30].

[xxiv] **Evolving notation** —
Bird took the work forwards through the 1980's, notably in a series of tutorial papers [31–33] produced in quick succession; an example, the calculation for the Maximum Segment Sum problem, is shown in Fig. 3.

[xxv] **The names "Squiggol" and "BMF"** —
Meertens recalls that Robert Dewar passed a note to him with the single word "Squigol" on it, making a pun with language names such as ALGOL, COBOL, and SNOBOL [138]. The first appearance of the name in the minutes is for Meeting #35 in Sausalito in December 1985. However, it has come to be written "Squiggol", perhaps to emphasise that the pronunciation should be ˈskwɪgɒl ("qui") rather than ˈskwaɪgɒl ("quae"). Later, at a meeting of the STOP project in Nijmegen in 1988, Doaitse Swierstra coined the more sober name "Bird–Meertens Formalism" (BMF), making a different pun with "Backus–Naur Form" (BNF).

[xxvi] **The Eindhoven quantifier notation** —
The Eindhoven quantifier notation rationalised the notation for binding a variable, determining its range and forming elements from it [11,153]. In the conventional $\sum_{n=0}^{N} n^2$ for example, the n below the $\sum$ is a binding occurrence; but the n in n^2 is bound; and the n^2 forms elements from that bound variable. The 0 and the N determine the range of n, and the $\sum$ itself gives the 'quantifier', the operation (usually associative and commutative) carried out on the elements. In the Eindhoven notation that would be written in the order quantifier, bound variable(s), range, element-former. The whole expression is *always* enclosed by binding-scope delimiters — so the example above might be written $(+n : 0 \leq n \leq N : n^2)$.
The advantage of using the Eindhoven notation is that uniform calculational laws apply to the manipulation of those expressions, and they greatly reduce the risk of error.

[xxvii] **Catamorphisms** —
Meertens coined the term catamorphism for the unique function induced by a homomorphism from the initial algebra, in a working document presented at Meeting #38 in Rome (1988).

[xxviii] **Datatype-generic programming** —
The term 'datatype-generic programming' was coined by Roland Backhouse and Jeremy Gibbons for a project that ran 2003–2006 [14]; the point was to distinguish from the different use of the term 'generic programming' in languages like C++, where it essentially means parametric polymorphism. Within the context of the Group, 'datatype-generic programming' has come to mean parametrization by a functor, as with catamorphisms, and plain 'generic programming' to mean functions defined more specifically over the sum-of-products structure of a polynomial functor, as with PolyP and Generic Haskell.

[xxix] **Polytypic programming languages and PolyP** —
The language PolyP, an extension of the lazy, higher-order functional programming language Haskell [173], was designed by Jansson and Jeuring at Chalmers, Gothenburg [111]. The development of PolyP and its applications was discussed at Meeting #49 in Rancho Santa Fe (1996), Meeting #51 in Oxford (1998), and Meeting #53 in Potsdam (1999).

[xxx] **Generic datatypes with mutual recursion** —
The theory to make Generic Haskell possible was developed by Hinze, a first-time observer in Potsdam (1999). He presented his theory at Meeting #54 in Blackheath (2000) [91]. To support generic functions on sets of mutually recursive datatypes, Hinze, Jeuring, and Löh developed Generic Haskell from 2000 onwards [94,119]. Various aspects of Generic Haskell were discussed also at Meeting #59 in Nottingham in 2004.

[xxxi] **Type-indexed datatypes** —
Type-indexed datatypes were introduced by Hinze et al. [94]. The type families extension of Haskell is based on the work of Chakravarty et al. [50].

[xxxii] **Fixed-point representation of mutually recursive datatypes** —
Rodriguez and others developed MultiRec [178], a generic programming library that uses a fixed-point representation of possibly mutually recursive datatypes.

[xxxiii] **Generic programming libraries** —
For an early comparison of generic programming libraries, see Rodriguez et al. [177]. An early variant of Scrap Your Boilerplate [118] was discussed at Meeting #56 on Ameland, The Netherlands (2001). Generic Deriving [122] was discussed at Meeting #70 in Ulm.

[xxxiv] **Catamorphisms** —
This work was done mainly by Grant Malcolm [126].

[xxxv] **A zoo of morphisms** —
There were mutumorphisms [71], which are pairs of mutually recursive functions; zygomorphisms [125], which consist of a main recursive function and an auxiliary one on which it depends; histomorphisms [195], in which the body has access to the recursive images of all subterms, not just the immediate ones; so-called generalised folds [28], which use polymorphic recursion to handle nested datatypes; and then there were generic accumulations [163], which keep intermediate results in additional paramters for later stages in the computation.

[xxxvi] **Paramorphism** —
This was introduced by Lambert Meertens at Meeting #41 in Burton, UK (1990) [139].

[xxxvii] **Recursion schemes from comonads** —
This appeared in Uustalu et al [197]. Comonads capture the general idea of 'evaluation in context' [196], and this scheme makes contextual information available to the body of the recursion. It was used to subsume both zygomorphisms and histomorphisms.

[xxxviii] **Adjoint folds** —
This was done by Hinze [92]. Using adjunctions as the common thread, adjoint folds arise by inserting a left adjoint functor into the recursive characterisation, thereby adapting the form of the recursion; they subsume paramorphisms, accumulating folds, mutumorphisms (and hence zygomorphisms), and generalised folds. Later, it was observed that adjoint folds could be used to subsume recursion schemes from comonads by Hinze and Wu [93].

[xxxix] **Constructive mathematics and computer programming** —
The connection between constructive mathematics and computer programming was pioneered by the Swedish philosopher and logician Per Martin-Löf [130].

[xl] **Programming languages implementing dependent types** —
Programming languages with dependent types include ALF [124], Cayenne [7], ATS [203], Epigram [132], Agda [159] and Idris [44].

[xli] **Curry-Howard correspondence** —
The Curry-Howard correspondence describes how the typing rules of the lambda calculus are in one-to-one correspondence with the natural deduction rules in logic. Wadler [200] gives a historic overview of this idea, aimed at a more general audience.

[xlii] **Generic programming in dependently typed languages** —
The idea of using dependent types to define an explicit *universe* of types was one of the early applications of dependently typed programming [4, 27]. Since then, there have been several PhD theses exploring this idea further [53,57,115,123,155,159]

[xliii] **WG2.1 and dependent types** —
Backhouse started exploring type theory in the mid 1980's [13]. At Meeting #42, Nordström was invited as an observer and talked about the work on ALF. Throughout the early 21st century, observers and members were frequently active in the area of type theory or generic programming, including McBride, Löh, Jansson, Swierstra, Dagand, McKinna and many others.

[xliv] **Algebra of programming in Agda** —
Patrik Jansson gave a first tutorial on the dependently typed programming language Agda at Meeting #63 in Kyoto in 2007. This lead to an exploration of how to mechanize the kind of program that was previously carried out on paper [156].

[xlv] **Program calculation and type theory** —
As type theory is a language for describing both proofs and programs, it is no surprise that it provides the ideal setting for formalizing the program calculation techniques that members of the Group pioneered [3, 190,192].

[xlvi] **Applications of dependent types** —
As languages with dependent types matured, various researchers started exploring novel and unexpected applications in a variety of domains [40, 58,109,110].

[xlvii] **Dependently typed combinator parsing** —
This was for example investigated by Nils Danielsson [58].

[xlviii] **Dependent types and program development** —
Many modern dependently typed programming languages are equipped with some sort of IDE. Once the type signature of a method has been fixed, the programmer can interactively find a suitable definition. There are numerous examples of how a powerful type signature can give strong guarantees about a data structure's invariants [131], the correctness of a domain-specific language [59], or type preservation of a compiler [134].

[xlix] **Dependent types and effects** —
There is a large body of work studying how to incorporate side-effects in dependently typed programming languages. This can be done by constructing denotational models [189, 191], by adding new effectful primitives to the type theory [157], or by giving an algebraic account of the properties that laws that effects satisfy [45, 77].

[l] **Monads** —
This insight was made by Eugenio Moggi while studying semantics of programming languages [141].

[li] **Kleisli arrows** —
Mike Spivey adopted this notion of monads for writing purely functional programs with exceptions [186]; Phil Wadler generalized it to other effects, and popularized it as the main abstraction for dealing with effects in Haskell [198, 199].

[lii] **Parser combinators** —
The combinator style of parsing is due to William Burge [48]. The monadic presentation was popularized by Graham Hutton and Erik Meijer [107], and a dependently typed version presented by Nils Danielsson [xlvii].

[liii] **Parsers in verse** —
The verse characterization of the parser type is due Fritz Ruehr [179].

[liv] **Applicative functors** —
The applicative interface for parsers was invented by Doaitse Swierstra [188]. This and other applications inspired Conor McBride and Ross Paterson to identify the abstraction of *applicative functors* (also called "strong lax-monoidal functors" or "idioms") [133]. Like monads, applicative functors have turned out to have unforeseen applications, such as in datatype traversals [29, 78] and distributed computing [75].

[lv] **Algebraic effects** —
Purely functional specifications of effects were studied by Wouter Swierstra in his PhD thesis [189, 191]. The axioms of an algebraic specification can be applied to equational reasoning involving either combinators or the imperative-flavoured comprehension notation provided for example by Haskell's **do** notation [77]. Algebraic effects and handlers were introduced by Gordon Plotkin then explored more fully in Matija Pretnar's PhD thesis [175], and are now the subject of much active work in the Group and beyond.

[lvi] **Applications of relation algebra** —
Roland Backhouse and B.A. Carré discovered similarities between an algebra for path problems and the algebra of regular languages [15]. Tony Hoare and others developed algebraic laws of programming, insisting that "specifications obey all the laws of the calculus of relations" [97]. Richard Bird and Oege de Moor used relations for the calculational derivation of programs covering principles of algorithm design such as dynamic programming, greedy algorithms, exhaustive search and divide and conquer [35].

[lvii] **Algebra of binary relations** —
Calculi based on the algebra of binary relations were developed by George Boole, Charles Peirce, Ernst Schröder, Augustus De Morgan and Alfred Tarski [171,180,194].

[lviii] **Graph algorithms** —
Walter Guttmann, for example, showed that the same correctness proof shows that well-known algorithms solve the minimum weight spanning tree problem, the minimum bottleneck spanning tree problem and similar optimisation problems with different aggregation functions [84]. Algebraic versions of Dijkstra's shortest path algorithm and the one by Floyd/Warshall are applications of these algorithms to structures different from graphs, pinpointing the mathematical requirements on the underlying cost algebra that ensure their correctness [102]. Roland Backhouse and colleagues are currently writing a book on algorithmic graph theory presented relationally [18].

[lix] **Program analysis** —
Program analysis using an algebraic style of reasoning has always been a core activity of the Group; for examples see [62,63,66].

[lx] **Pointer structures** —
Bernhard Möller and Richard Bird researched representations of data structures in general, and pointer structures in particular [34,142].

[lxi] **Algebraic logics** —
An important step to an algebraic form of program logic was taken by Hoare and his colleagues [97]. More recently, the central aspects of Separation Logic [160,161] were treated algebraically [54–56].
Next to programming semantics, the infinite iteration operator can be applied to model various logics. The temporal logics LTL, CTL and CTL* have been in [60,114,150]. There were studies on logics for hybrid systems [100,101] and Neighbourhood Logic [99].

[lxii] **Further applications of the algebraic approach** —
The Group discovered countless areas in computer science where semirings are the underlying structure. Applications reach from, fundamental concepts of programming language semantics, including concurrent programs [98] and termination [16,61,66,90] via games [12,17,183] and data processing [174], to multi-agent systems [144] and quantum computing [193].
Beyond that, matrix-style reasoning has applications in object-oriented

programming [121] and feature-oriented software development, including aspects of product families [106] and of feature interactions [20].

[lxiii] **Algebraic semantics of the while loop** —
The fixed-point characterisation of while loops goes back to Andrzej Blikle and David Park [37,164]. Dexter Kozen transferred the concept into the setting of Kleene algebras [116].

[lxiv] **Algebras with tests** —
Test elements form a Boolean subalgebra. It represents an algebraic version of the usual assertion logics like the Hoare calculus [117,147]. There is a direct link to weakest (liberal) preconditions [35,148].

[lxv] **Fixed-point fusion** —
Fixed-point fusion is a consequence of the fixed-point semantics of recursion [1,140].

[lxvi] **Virtual data structures** —
These were described by Doaitse Swierstra and Oege de Moor [187].

[lxvii] **Omega algebras** —
The omega operator was introduce by Cohen [51]; Möller performed a systematic study of its foundations [143].
Guttmann used it for analysing executions of lazy and strict computations [82]. Infinite traces, also called *streams*, have many applications including the modelling protocols [142], as well as dynamic and hybrid systems [100,183,184] . The corresponding algebras can also be used to formally reason about (non)termination in classical programs [104].

[lxviii] **Tool-Support for algebraic reasoning** —
Peter Höfner and Georg Struth proved countless theorems of all these algebras in automated theorem provers, such as Prover9 [103,105]. Walter Guttmann, Peter Höfner, Georg Struth and others used the interactive proof assistant Isabelle/HOL to implement the algebras, the concrete models, as well as many program derivations, e.g. [5,6,80,83].

[lxix] **Program transformation** —
In the functional realm, fundamental ideas in program transformation were introduced by Cooper [52] and subsequently developed by others, in particular Burstall and Darlington [49]. Later activities occurred within the ISI project [19,120] and at Kestrel Institute [81]. In the realm of artificial intelligence there were ideas in the field of automated programming (e.g., the DEDALUS system [127] and its successor [128,129]).

[lxx] **Refinement calculi** —
Imperative programming calculi based on refinement include those of Dijkstra [65], Back [8], Hoare [96,97], Hehner [87–89], Morris [154], and Morgan [151,152].

[lxxi] **Transformational development** —
For background on the 'life cycle of transformational program development', see Broy [2]. The five levels of the 'wide spectrum' are due to Partsch [168].

[lxxii] **The language CIP-L** —
The language CIP-L is described in detail in the first of two volumes about the CIP project as a whole [24]. For some of the motivation, see Bauer [22] and Broy and Pepper [47].

[lxxiii] **The system CIP-S** —
The specification of the CIP-S system can be found in the second volume about the CIP project [25]. The more interesting parts of the formal development of the system, together with the transformation rules used, can also be found there. Successors to CIP-S were developed by Partsch [169] and Guttmann *et al.* [85].

[lxxiv] **Experiences with CIP** —
Smaller CIP case studies include sorting [46,165] and parsing [166–168]. As noted above, the CIP-S system itself [25] constitutes a larger case study.

[lxxv] **Programs should be grown** —
Fred Brooks wrote "Some years ago Harlan Mills proposed that any software system should be grown by incremental development." [72]

Acknowledgements. Section 2 is based on a paper more specifically about the evolution of the Bird–Meertens Formalism [76], Sect. 3 partly based on a paper about the contributions to generic programming of the Software Technology group at Utrecht University [112], and Sect. 4 partly based on a paper about the unification of recursion schemes [93].

References

1. Aarts, C., et al.: Fixed-point calculus. Inf. Process. Lett. **53**(3), 131–136 (1995)
2. Agresti, W.M.: What are the new paradigms? In: Agresti, W.M. (ed.) New Paradigms for Software Development. IEEE Computer Society Press (1986)
3. Alpuim, J., Swierstra, W.: Embedding the refinement calculus in Coq. Sci. Comput. Program. **164**, 37–48 (2018)
4. Altenkirch, T., Mcbride, C.: Generic programming within dependently typed programming. In: Gibbons, J., Jeuring, J. (eds.) Generic Programming. ITIFIP, vol. 115, pp. 1–20. Springer, Boston, MA (2003). https://doi.org/10.1007/978-0-387-35672-3_1
5. Armstrong, A., Struth, G., Weber, T.: Kleene algebra. Archive of Formal Proofs (2013). http://isa-afp.org/entries/Kleene_Algebra.html
6. Armstrong, A., Foster, S., Struth, G., Weber, T.: Relation algebra. Archive of Formal Proofs (2014). http://isa-afp.org/entries/Relation_Algebra.html
7. Augustsson, L.: Cayenne - a language with dependent types. In: International Conference on Functional Programming, ICFP 1998, pp. 239–250 (1998)
8. Back, R.J.: On the correctness of refinement steps in program development. PhD thesis. Report A-1978-4, Department of Computer Science, University of Helsinki (1978)
9. Back, R.J.: On correct refinement of programs. J. Comput. Syst. Sci. **23**(1), 49–68 (1981). https://doi.org/10.1016/0022-0000(81)90005-2

10. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science, Springer (1998). https://doi.org/10.1007/978-1-4612-1674-2_4
11. Backhouse, R.: An exploration of the Bird-Meertens formalism. Technical report CS 8810, Department of Computer Science, Groningen University (1988)
12. Backhouse, R., Michaelis, D.: Fixed-point characterisation of winning strategies in impartial games. In: Berghammer, R., Möller, B., Struth, G. (eds.) Relational and Kleene-Algebraic Methods in Computer Science. Lecture Notes in Computer Science, vol. 3051, pp. 34–47. Springer (2004)
13. Backhouse, R., Chisholm, P., Malcolm, G., Saaman, E.: Do-it-yourself type theory. Formal Aspects Comput. **1**(1), 19–84 (1989)
14. Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.): Spring School on Datatype-Generic Programming, Lecture Notes in Computer Science, vol. 4719. Springer-Verlag (2007). https://doi.org/10.1007/978-3-540-76786-2
15. Backhouse, R.C., Carré, B.A.: Regular algebra applied to path-finding problems. IMA J. Appl. Math. **15**(2), 161–186 (1975). https://doi.org/10.1093/imamat/15.2.161
16. Backhouse, R.C., Doornbos, H.: Datatype-generic termination proofs. Theor. Comput. Syst. **43**(3–4), 362–393 (2008). https://doi.org/10.1007/s00224-007-9056-z
17. Backhouse, R.C., Chen, W., Ferreira, J.F.: The algorithmics of solitaire-like games. Sci. Comput. Program. **78**(11), 2029–2046 (2013). https://doi.org/10.1016/j.scico.2012.07.007
18. Backhouse, R.C., Doornbos, H., Glück, R., van der Woude, J.: Elements of algorithmic graph theory: an exercise in point-free reasoning, (working document) (2019)
19. Balzer, R., Goldman, N., Wile, D.: On the transformational implementation approach to programming. In: Yeh, R.T., Ramamoorthy, C.V. (eds.) International Conference on Software Engineering, IEEE Computer Society, pp. 337–344 (1976)
20. Batory, D.S., Höfner, P., Kim, J.: Feature interactions, products, and composition. In: Denney, E., Schultz, U.P. (eds.) Generative Programming and Component Engineering. ACM, pp. 13–22 (2011). https://doi.org/10.1145/2047862.2047867
21. Bauer, F.L.: Programming as an evolutionary process. In: Yeh, R.T., Ramamoorthy, C. (eds.) International Conference on Software Engineering, IEEE Computer Society, pp. 223–234 (1976)
22. Bauer, F.L.: From specifications to machine code: Program construction through formal reasoning. In: Ohno, Y., Basili, V., Enomoto, H., Kobayashi, K., Yeh, R.T. (eds.) International Conference on Software Engineering, IEEE Computer Society, pp. 84–91 (1982)
23. Bauer, F.L., Wössner, H.: Algorithmic Language and Program Development. Texts and Monographs in Computer Science. Springer (1982). https://doi.org/10.1007/978-3-642-61807-9
24. Bauer, F.L., et al.: The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L. Lecture Notes in Computer Science, vol. 183. Springer (1985). https://doi.org/10.1007/3-540-15187-7
25. Bauer, F.L., et al.: The Munich Project CIP, Volume II: The Program Transformation System CIP-S, Lecture Notes in Computer Science, vol. 292. Springer-Verlag, Berlin (1987). https://doi.org/10.1007/3-540-18779-0
26. Bemer, R.: A politico-social history of ALGOL. In: Annual Review of Automatic Programming 5, pp. 151–237. Pergamon Press, Oxford (1969)

27. Benke, M., Dybjer, P., Jansson, P.: Universes for generic programs and proofs in dependent type theory. Nordic J. Comput. **10**(4), 265–289 (2003)
28. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects Comput. **11**(2), 200–222 (1999). https://doi.org/10.1007/s001650050047
29. Bird, R., Gibbons, J., Mehner, S., Voigtländer, J., Schrijvers, T.: Understanding idiomatic traversals backwards and forwards. In: Haskell Symposium. ACM (2013). https://doi.org/10.1145/25037782503781 (2013)
30. Bird, R.S.: Some notational suggestions for transformational programming. Working Paper NIJ-3, IFIP WG2.1, also Technical Report RCS 144, Department of Computer Science, University of Reading (1981)
31. Bird, R.S.: An introduction to the theory of lists. Monograph PRG-56, Programming Research Group, University of Oxford (1986)
32. Bird, R.S.: A calculus of functions for program derivation. Monograph PRG-64, Programming Research Group, University of Oxford (1987)
33. Bird, R.S.: Lectures on constructive functional programming. Monograph PRG-69, Programming Research Group, University of Oxford (1988)
34. Bird, R.S.: Unfolding pointer algorithms. J. Funct. Program. **11**(3), 347–358 (2001). https://doi.org/10.1017/S0956796801003914
35. Bird, R.S., de Moor, O.: Algebra of Programming. Prentice Hall International Series in Computer Science. Prentice Hall, Hoboken (1997)
36. Blaine, L., Gilham, L., Liu, J., Smith, D.R., Westfold, S.J.: Planware: domain-specific synthesis of high-performance schedulers. In: Automated Software Engineering, IEEE Computer Society, p. 270 (1998). https://doi.org/10.1109/ASE.1998.732672
37. Blikle, A.: Iterative systems: An algebraic approach. Bulletin de l'Académie Polonaise des Sciences, Série des sciences mathématiques, astronomiques et physiques XX(1) (1972)
38. Boom, H.: Further thoughts on Abstracto. Working Paper ELC-9, IFIP WG2.1 (1981)
39. Botta, N., Jansson, P., Ionescu, C.: Contributions to a computational theory of policy advice and avoidability. J. Funct. Programm. **27**, e23 (2017) . https://doi.org/10.1017/S0956796817000156
40. Botta, N., Jansson, P., Ionescu, C., Christiansen, D.R., Brady, E.: Sequential decision problems, dependent types and generic solutions. Logical Meth. Comput. Sci. **13**(1) (2017). https://doi.org/10.23638/LMCS-13(1:7)2017
41. Boyle, J., Harmer, T.J., Winter, V.L.: The TAMPR program transformation system: simplifying the development of numerical software. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds,) Modern Software Tools for Scientific Computing, Birkhäuser, pp. 353–372 (1996) . https://doi.org/10.1007/978-1-4612-1986-6_17
42. Boyle, J.M.: An introduction to Transformation-Assisted Multiple Program Realization (TAMPR) system. In: Bunch, J.R. (ed.) Cooperative Development of Mathematical Software, Department of Mathematics, University of California, San Diego (1976)
43. Boyle, J.M., Dritz, K.W.: An automated programming system to facilitate the development of quality mathematical software. In: Rosenfeld, J. (ed.) IFIP Congress, North-Holland, pp. 542–546 (1974)
44. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. J. Funct. Program. **23**(5), 552–593 (2013)
45. Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: International Conference on Functional Programming, pp. 133–144 (2013)

46. Broy, M.: Program construction by transformations: a family tree of sorting programs. In: Biermann, A., Guiho, G. (eds.) Computer Program Synthesis Methodologies, NATO Advanced Study Institutes Series, vol. 95. Springer (1983). https://doi.org/10.1007/978-94-009-7019-9_1
47. Broy, M., Pepper, P.: On the coherence of programming language and programming methodology. In: Bormann, (ed.) IFIP Working Conference on Programming Languages and System Design, North-Holland, pp. 41–53 (1983)
48. Burge, W.H.: Recursive Programming Techniques. Addison-Wesley, Boston (1975)
49. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977)
50. Chakravarty, M.M.T., Keller, G., Jones, S.L.P., Marlow, S.: Associated types with class. In: Palsberg, J., Abadi, M. (eds.) Principles of Programming Languages. ACM, pp. 1–13 (2005). https://doi.org/10.1145/1040305.1040306
51. Cohen, E.: Separation and reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000). https://doi.org/10.1007/10722010_4
52. Cooper, D.: The equivalence of certain computations. Comput. J. **9**, 45–52 (1966)
53. Dagand, P.E., et al.: A cosmology of datatypes: Reusability and dependent types. Ph.D. thesis, University of Strathclyde (2013)
54. Dang, H., Möller, B.: Concurrency and local reasoning under reverse exchange. Sci. Comput. Programm. **85**, Part B, 204–223 (2013)
55. Dang, H., Möller, B.: Extended transitive separation logic. J. Logical Algebraic Meth. Programm. **84**(3), 303–325 (2015). https://doi.org/10.1016/j.jlamp.2014.12.002
56. Dang, H., Höfner, P., Möller, B.: Algebraic separation logic. J. Logic Algebraic Programm. **80**(6), 221–247 (2011). https://doi.org/10.1016/j.jlap.2011.04.003
57. Danielsson, N.A.: Functional program correctness through types. Ph.D. thesis, Chalmers University of Technology and Gothenburg University (2007)
58. Danielsson, N.A.: Total parser combinators. In: International Conference on Functional Programming, pp. 285–296 (2010)
59. Danielsson, N.A.: Correct-by-construction pretty-printing. In: Workshop on Dependently-Typed Programming, pp. 1–12 (2013)
60. Desharnais, J., Möller, B.: Non-associative Kleene algebra and temporal logics. In: Höfner, P., Pous, D., Struth, G. (eds.) Relational and Algebraic Methods in Computer Science. Lecture Notes in Computer Science, vol. 10226, pp. 93–108 (2017). https://doi.org/10.1007/978-3-319-57418-9_6
61. Desharnais, J., Möller, B., Struth, G.: Termination in modal Kleene algebra. In: Mayr, E.W., Mitchell, J.C., Lévy, J.J. (eds.) Exploring New Frontiers of Theoretical Informatics, pp. 647–660, Kluwer (2004)
62. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. ACM Trans. Comput. Log. **7**(4), 798–833 (2006)
63. Desharnais, J., Möller, B., Tchier, F.: Kleene under a modal demonic star. J. Logic Algebraic Programm. **66**(2), 127–160 (2006). https://doi.org/10.1016/j.jlap.2005.04.006
64. Dewar, R.: Letter to members of IFIP WG2.1 (1977). http://ershov-arc.iis.nsk.su/archive/eaindex.asp?did=29067
65. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Hoboken (1976)
66. Doornbos, H., Backhouse, R.C.: Algebra of program termination. In: Backhouse, R.C., Crole, R.L., Gibbons, J. (eds.) Algebraic and Coalgebraic Methods in the

Mathematics of Program Construction, Lecture Notes in Computer Science, vol. 2297, pp. 203–236. Springer (2000). https://doi.org/10.1007/3-540-47797-7_6
67. Feather, M.S.: A system for developing programs by transformation. Ph.D thesis, University of Edinburgh, UK (1979). http://hdl.handle.net/1842/7296
68. Feather, M.S.: A system for assisting program transformation. ACM Trans. Programm. Lang. **4**(1), 1–20 (1982). https://doi.org/10.1145/357153.357154
69. Feather, M.S.: A survey and classification of some program transformation approaches and techniques. In: Meertens, L. (ed.) Program Specification and Transformation, North-Holland, pp. 165–195 (1987)
70. Floyd, R.W.: Assigning meaning to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science, American Mathematical Society, Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 19–32 (1967)
71. Fokkinga, M.: Tupling and mutumorphisms. The Squiggolist **1**(4), 81–82 (1990)
72. Brooks, J.F.: The Mythical Man-Month. Addison-Wesley, Boston (1975)
73. Geurts, L., Meertens, L.: Remarks on Abstracto. Algol. Bull. **42**, 56–63 (1978)
74. Geurts, L., Meertens, L., Pemberton, S.: The ABC Programmer's Handbook. Prentice-Hall, Hoboken, iSBN 0-13-000027-2 (1990)
75. Gibbons, J.: Free delivery (functional pearl). In: Haskell Symposium, pp. 45–50 (2016). https://doi.org/10.1145/2976002.2976005
76. Gibbons, J.: The school of Squiggol: A history of the Bird-Meertens formalism. In: Astarte, T. (ed.) Workshop on the History of Formal Methods. Springer-Verlag, Lecture Notes in Computer Science (2020). (to appear)
77. Gibbons, J., Hinze, R.: Just do it: Simple monadic equational reasoning. In: International Conference on Functional Programming, pp. 2–14 (2011). https://doi.org/10.1145/2034773.2034777
78. Gibbons, J., dos Santos Oliveira, B.C.: The essence of the iterator pattern. J. Funct. Programm. **19**(3,4), 377–402 (2009). https://doi.org/10.1017/S0956796809007291
79. Gibbons, J., Henglein, F., Hinze, R., Wu, N.: Relational algebra by way of adjunctions. Proc. ACM Programm. Lang. **2**(ICFP), 86:1–86:28 (2018). https://doi.org/10.1145/3236781
80. Gomes, V.B.F., Guttmann, W., Höfner, P., Struth, G., Weber, T.: Kleene algebras with domain. Archive of Formal Proofs (2016). http://isa-afp.org/entries/KAD.html
81. Green, C., et al.: Research on knowledge-based programming and algorithm design. Technical report Kes.U.81.2, Kestrel Institute (1981, revised 1982) (1981)
82. Guttmann, W.: Infinite executions of lazy and strict computations. J. Logical Algebraic Meth. Programm. **84**(3), 326–340 (2015). https://doi.org/10.1016/j.jlamp.2014.08.001
83. Guttmann, W.: Stone algebras. Archive of Formal Proofs (2016). http://isa-afp.org/entries/Stone_Algebras.html
84. Guttmann, W.: An algebraic framework for minimum spanning tree problems. Theoret. Comput. Sci. **744**, 37–55 (2018)
85. Guttmann, W., Partsch, H., Schulte, W., Vullinghs, T.: Tool support for the interactive derivation of formally correct functional programs. J. Univ. Comput. Sci. **9**(2), 173 (2003). https://doi.org/10.3217/jucs-009-02-0173
86. Hagino, T.: A categorical programming language. Ph.D thesis, University of Edinburgh, UK (1987)
87. Hehner, E.C.R.: Predicative programming, part I. Commun. ACM **27**(2), 134–143 (1984). https://doi.org/10.1145/69610.357988

88. Hehner, E.C.R.: Predicative programming, part II. Commun. ACM **27**(2), 144–151 (1984). https://doi.org/10.1145/69610.357990
89. Hehner, E.C.R.: A Practical Theory of Programming. Springer (1993). https://doi.org/10.1007/978-1-4419-8596-5_7
90. Hehner, E.C.R.: Specifications, programs, and total correctness. Sci. Comput. Program. **34**(3), 191–205 (1999). https://doi.org/10.1016/S0167-6423(98)00027-6
91. Hinze, R.: Polytypic values possess polykinded types. Sci. Comput. Program. **43**(2–3), 129–159 (2002)
92. Hinze, R.: Adjoint folds and unfolds–an extended study. Sci. Comput. Program. **78**(11), 2108–2159 (2013). https://doi.org/10.1016/j.scico.2012.07.011
93. Hinze, R., Wu, N.: Unifying structured recursion schemes: an extended study. J. Funct. Program. **26**, 47 (2016)
94. Hinze, R., Jeuring, J., Löh, A.: Type-indexed data types. Sci. Comput. Program. **51**(1–2), 117–151 (2004)
95. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
96. Hoare, C.A.R.: Programs are predicates. Philosophical Transactions of the Royal Society of London (A 312), 475–489 (1984)
97. Hoare, C.A.R., et al.: Laws of programming. Commun. ACM **30**(8), 672–686 (1987). https://doi.org/10.1145/27651.27653
98. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. J. Logic Algebraic Programm. **80**(6), 266–296 (2011). https://doi.org/10.1016/j.jlap.2011.04.005
99. Höfner, P., Möller, B.: Algebraic neighbourhood logic. J. Logic Algebraic Programm. **76**, 35–59 (2008)
100. Höfner, P., Möller, B.: An algebra of hybrid systems. J. Logic Algebraic Programm. **78**, 74–97 (2009). https://doi.org/10.1016/j.jlap.2008.08.005
101. Höfner, P., Möller, B.: Fixing Zeno gaps. Theoret. Comput. Sci. **412**(28), 3303–3322 (2011). https://doi.org/10.1016/j.tcs.2011.03.018
102. Höfner, P., Möller, B.: Dijkstra, Floyd and Warshall meet Kleene. Formal Aspects Comput. **24**(4–6), 459–476 (2012). https://doi.org/10.1007/s00165-012-0245-4
103. Höfner, P., Struth, G.: Automated reasoning in Kleene algebra. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 279–294. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_19
104. Höfner, P., Struth, G.: Non-termination in idempotent semirings. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS 2008. LNCS, vol. 4988, pp. 206–220. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78913-0_16
105. Höfner, P., Struth, G.: On automating the calculus of relations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 50–66. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_5
106. Höfner, P., Khédri, R., Möller, B.: Supplementing product families with behaviour. Softw. Inform. **5**(1–2), 245–266 (2011)
107. Hutton, G., Meijer, E.: Monadic parsing in Haskell. J. Funct. Program. **8**(4), 437–444 (1998). https://doi.org/10.1017/S0956796898003050
108. Ionescu, C.: Vulnerability modelling with functional programming and dependent types. Math. Struct. Comput. Sci. **26**(1), 114–128 (2016). https://doi.org/10.1017/S0960129514000139
109. Ionescu, C., Jansson, P.: Dependently-typed programming in scientific computing. In: Hinze, R. (ed.) IFL 2012. LNCS, vol. 8241, pp. 140–156. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41582-1_9

110. Ionescu, C., Jansson, P.: Testing versus proving in climate impact research. In: TYPES 2011, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Leibniz International Proceedings in Informatics (LIPIcs), vol. 19, pp. 41–54 (2013). https://doi.org/10.4230/LIPIcs.TYPES.2011.41
111. Jansson, P., Jeuring, J.: PolyP – a polytypic programming language extension. In: Principles of Programming Languages, pp. 470–482 (1997)
112. Jeuring, J., Meertens, L.: Geniaal programmeren-generic programming at Utrecht-. In: et al. HB (ed.) Fascination for computation, 25 jaar opleiding informatica, Department of Information and Computing Sciences, Utrecht University, pp. 75–88 (2009)
113. Johnson, W.L., Feather, M.S., Harris, D.R.: The KBSA requirements/specifications facet: ARIES. In: Knowledge-Based Software Engineering, IEEE Computer Society, pp. 48–56 (1991). https://doi.org/10.1109/KBSE.1991.638020
114. von Karger, B., Berghammer, R.: A relational model for temporal logic. Logic J. IGPL **6**, 157–173 (1998)
115. Ko, H.S.: Analysis and synthesis of inductive families. DPhil thesis, Oxford University, UK (2014)
116. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (1997)
117. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Trans. Comput. Log. **1**(1), 60–76 (2000)
118. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical design pattern for generic programming. In: Types in Language Design and Implementation, pp. 26–37 (2003)
119. Löh, A., Clarke, D., Jeuring, J.: Dependency-style generic Haskell. In: Shivers, O. (ed.) International Conference on Functional Programming. ACM Press, pp. 141–152 (2003)
120. London, P., Feather, M.: Implementing specification freedoms. Sci. Comput. Program. **2**(2), 91–131 (1982)
121. Macedo, H., Oliveira, J.N.: A linear algebra approach to OLAP. Formal Aspects Comput. **27**(2), 283–307 (2015). https://doi.org/10.1007/s00165-014-0316-9
122. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for Haskell. In: Haskell Symposium, pp. 37–48 (2010)
123. Magalhães, J.P.R.: Less is more: generic programming theory and practice. PhD thesis, Utrecht University, Netherlands (2012)
124. Magnusson, L., Nordström, B.: The ALF proof editor and its proof engine. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 213–237. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58085-9_78
125. Malcolm, G.: Algebraic data types and program transformation. PhD thesis, University of Groningen (1990)
126. Malcolm, G.: Data structures and program transformation. Sci. Comput. Program. **14**, 255–279 (1990)
127. Manna, Z., Waldinger, R.J.: Synthesis: dreams → programs. IEEE Trans. Software Eng. **5**(4), 294–328 (1979). https://doi.org/10.1109/TSE.1979.234198
128. Manna, Z., Waldinger, R.J.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980). https://doi.org/10.1145/357084.357090
129. Manna, Z., Waldinger, R.J.: The Deductive Foundations of Computer Programming. Addison-Wesley, Boston (1993)

130. Martin-Löf, P.: Constructive mathematics and computer programming. In: Studies in Logic and the Foundations of Mathematics, vol. 104, Elsevier, pp. 153–175 (1982)
131. McBride, C.: How to keep your neighbours in order. In: International Conference on Functional Programming, Association for Computing Machinery, New York, NY, USA, ICFP 2014, pp. 297–309 (2014). https://doi.org/10.1145/2628136.2628163
132. McBride, C., McKinna, J.: The view from the left. J. Funct. Program. **14**(1), 69–111 (2004)
133. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (2008). https://doi.org/10.1017/S0956796807006326
134. McKinna, J., Wright, J.: A type-correct, stack-safe, provably correct, expression compiler in Epigram, unpublished draft (2006)
135. Meertens, L.: Abstracto 84: The next generation. In: Proceedings of the 1979 Annual Conference. ACM, pp. 33–39 (1979)
136. Meertens, L.: Algorithmics: Towards programming as a mathematical activity. In: de Bakker, J.W., Hazewinkel, M., Lenstra, J.K. (eds.) Proceedings of the CWI Symposium on Mathematics and Computer Science, North-Holland, pp. 289–334 (1986). https://ir.cwi.nl/pub/20634
137. Meertens, L.: An Abstracto reader prepared for IFIP WG 2.1. Technical report CS-N8702, CWI, Amsterdam (1987)
138. Meertens, L.: Squiggol versus Squigol, private email to JG (2019)
139. Meertens, L.G.L.T.: Paramorphisms. Formal Aspects Comput. **4**(5), 413–424 (1992)
140. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science, vol. 523. Springer, pp. 124–144 (1991). https://doi.org/10.1007/3540543961_7
141. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
142. Möller, B.: Calculating with pointer structures. In: IFIP TC2/WG 2.1 Working Conference on Algorithmic Languages and Calculi, pp. 24–48. Chapman & Hall (1997)
143. Möller, B.: Kleene getting lazy. Sci. Comput. Program. **65**, 195–214 (2007)
144. Möller, B.: Modal knowledge and game semirings. Comput. J. **56**(1), 53–69 (2013). https://doi.org/10.1093/comjnl/bxs140
145. Möller, B.: Geographic wayfinders and space-time algebra. J. Logical Algebraic Meth. Programm. **104**, 274–302 (2019). https://doi.org/10.1016/j.jlamp.2019.02.003
146. Möller, B., Roocks, P.: An algebra of database preferences. J. Logical Algebraic Meth. Programm. **84**(3), 456–481 (2015). https://doi.org/10.1016/j.jlamp.2015.01.001
147. Möller, B., Struth, G.: Modal Kleene algebra and partial correctness. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 379–393. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27815-3_30
148. Möller, B., Struth, G.: wp Is wlp. In: MacCaull, W., Winter, W., Düntsch, I. (eds.) Relational Methods in Computer Science. Lecture Notes in Computer Science, vol. 3929, pp. 200–211. Springer (2005). https://doi.org/10.1007/11734673_16
149. Möller, B., Partsch, H., Pepper, P.: Programming with transformations: an overview of the Munich CIP project (1983)

150. Möller, B., Höfner, P., Struth, G.: Quantales and temporal logics. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 263–277. Springer, Heidelberg (2006). https://doi.org/10.1007/11784180_21
151. Morgan, C.: The specification statement. ACM Trans. Program. Lang. Syst. **10**(3), 403–419 (1988). https://doi.org/10.1145/44501.44503
152. Morgan, C.: Programming from Specifications. Prentice Hall, Hoboken (1990)
153. Morgan, C.: An old new notation for elementary probability theory. Sci. Comput. Program. **85**, 115–136 (2014). https://doi.org/10.1016/j.scico.2013.09.003. special Issue on Mathematics of Program Construction 2012
154. Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. Sci. Comput. Program. **9**(3), 287–306 (1987)
155. Morris, P.W.: Constructing universes for generic programming. PhD thesis, University of Nottingham, UK (2007)
156. Mu, S.C., Ko, H.S., Jansson, P.: Algebra of programming in Agda: dependent types for relational program derivation. J. Funct. Program. **19**(5), 545–579 (2009)
157. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare type theory. In: International Conference on Functional Programming, pp. 62–73 (2006)
158. Naur, P.: The IFIP working group on ALGOL. ALGOL Bull. (Issue 15), 52 (1962)
159. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology (2007)
160. O'Hearn, P.: Resources, concurrency, and local reasoning. Theoret. Comput. Sci. **375**, 271–307 (2007)
161. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
162. Paige, R.: Transformational programming – Applications to algorithms and systems. In: Wright, J.R., Landweber, L., Demers, A.J., Teitelbaum, T. (eds.) Principles of Programming Languages. ACM, pp. 73–87 (1983). https://doi.org/10.1145/567067.567076
163. Pardo, A.: Generic accumulations. In: Gibbons, J., Jeuring, J. (eds.) Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming. Kluwer Academic Publishers, International Federation for Information Processing, vol. 115, pp. 49–78 (2002)
164. Park, D.: On the semantics of fair parallelism. In: Bjøorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 504–526. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10007-5_47
165. Partsch, H.: An exercise in the transformational derivation of an efficient program by joing development of control and data structure. Sci. Comput. Program. **3**(1), 1–35 (1983). https://doi.org/10.1016/0167-6423(83)90002-3
166. Partsch, H.: Structuring transformational developments: a case study based on Earley's recognizer. Sci. Comput. Program. **4**(1), 17–44 (1984). https://doi.org/10.1016/0167-6423(84)90010-8
167. Partsch, H.: Transformational derivation of parsing algorithms executable on parallel architectures. In: Ammann. U. (ed.) Programmiersprachen und Programmentwicklung, Informatik-Fachberichte, vol. 77, pp. 41–57. Springer (1984). https://doi.org/10.1007/978-3-642-69393-9_3
168. Partsch, H.: Transformational program development in a particular program domain. Sci. Comput. Program. **7**(2), 99–241 (1986). https://doi.org/10.1016/0167-6423(86)90008-0

169. Partsch, H.: Specification and Transformation of Programs – A Formal Approach to Software Development. Texts and Monographs in Computer Science. Springer (1990). https://doi.org/10.1007/978-3-642-61512-2
170. Partsch, H., Steinbrüggen, R.: Program transformation systems. ACM Comput. Surv. **15**(3), 199–236 (1983)
171. Peirce, C.S.: Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole's calculus of logic. Memoirs Am. Acad. Arts Sci. **9**, 317–378 (1870)
172. Pepper, P., Smith, D.R.: A high-level derivation of global search algorithms (with constraint propagation). Sci. Comput. Program. **28**(2–3), 247–271 (1997). https://doi.org/10.1016/S0167-6423(96)00023-8
173. Jones, S.P., et al.: Haskell 98, Language and Libraries. The Revised Report. Cambridge University Press, a special issue of the Journal of Functional Programming (2003)
174. Pontes, R., Matos, M., Oliveira, J.N., Pereira, J.O.: Implementing a linear algebra approach to data processing. In: Cunha, J., Fernandes, J.P., Lämmel, R., Saraiva, J., Zaytsev, V. (eds.) GTTSE 2015. LNCS, vol. 10223, pp. 215–222. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60074-1_9
175. Pretnar, M.: The logic and handling of algebraic effects. PhD thesis, School of Informatics, University of Edinburgh (2010)
176. Python Software Foundation: Python website (1997). https://www.python.org/
177. Yakushev, A.R., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O., Oliveira, B.C.D.S.: Comparing libraries for generic programming in Haskell. In: Haskell Symposium, pp. 111–122 (2008)
178. Yakushev, A.R., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. In: Hutton, G., Tolmach, A.P. (eds.) International Conference on Functional Programming, pp. 233–244 (2009)
179. Ruehr, F.: Dr Seuss on parser monads (2001). https://willamette.edu/~fruehr/haskell/seuss.html
180. Schröder, E.: Vorlesungen über die Algebra der Logik, vol 3. Taubner (1895)
181. Schuman, S.A. (ed.): New Directions in Algorithmic Languages, Prepared for IFIP Working Group 2.1 on Algol, Institut de Recherche d'Informatique et d'Automatique (1975)
182. Schuman, S.A. (ed.): New Directions in Algorithmic Languages, Prepared for IFIP Working Group 2.1 on Algol, Institut de Recherche d'Informatique et d'Automatique (1976)
183. Sintzoff, M.: On the design of correct and optimal dynamical systems and games. Inf. Process. Lett. **88**(1–2), 59–65 (2003). https://doi.org/10.1016/S0020-0190(03)00387-9
184. Sintzoff, M.: Synthesis of optimal control policies for some infinite-state transition systems. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 336–359. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70594-9_18
185. Smith, D.R.: KIDS: a semiautomatic program development system. IEEE Trans. Softw. Eng. **16**(9), 1024–1043 (1990). https://doi.org/10.1109/32.58788
186. Spivey, J.M.: A functional theory of exceptions. Sci. Comput. Program. **14**(1), 25–42 (1990). https://doi.org/10.1016/0167-6423(90)90056-J
187. Swierstra, S.D., de Moor, O.: Virtual data structures. In: Möller, B., Partsch, H., Schuman, S. (eds.) Formal Program Development. LNCS, vol. 755, pp. 355–371. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57499-9_26

188. Swierstra, S.D., Duponcheel, L.: Deterministic, error-correcting combinator parsers. In: Launchbury, J., Meijer, E., Sheard, T. (eds.) AFP 1996. LNCS, vol. 1129, pp. 184–207. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61628-4_7
189. Swierstra, W.: A functional specification of effects. PhD thesis, University of Nottingham (2008)
190. Swierstra, W., Alpuim, J.: From proposition to program. In: Kiselyov, O., King, A. (eds.) FLOPS 2016. LNCS, vol. 9613, pp. 29–44. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29604-3_3
191. Swierstra, W., Altenkirch, T.: Beauty in the beast. In: Haskell Workshop, pp. 25–36 (2007). http://doi.acm.org/10.1145/1291201.1291206
192. Swierstra, W., Baanen, T.: A predicate transformer semantics for effects (functional pearl). Proc. ACM Programm. Lang. **3**(ICFP), 1–26 (2019)
193. Tafliovich, A., Hehner, E.C.R.: Quantum predicative programming. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 433–454. Springer, Heidelberg (2006). https://doi.org/10.1007/11783596_25
194. Tarski, A.: On the calculus of relations. J. Symb. Log. **6**(3), 73–89 (1941). https://doi.org/10.2307/2268577
195. Uustalu, T., Vene, V.: Primitive (co)recursion and course-of-value (co)iteration, categorically. Informatica **10**(1), 5–26 (1999)
196. Uustalu, T., Vene, V.: Comonadic notions of computation. Electron. Notes Theoer. Comput. Sci. **203**(5), 263–284 (2008). https://doi.org/10.1016/j.entcs.2008.05.029
197. Uustalu, T., Vene, V., Pardo, A.: Recursion schemes from comonads. Nordic J. Comput. **8**(3), 366–390 (2001)
198. Wadler, P.: Comprehending monads. In: LISP and Functional Programming. ACM, pp. 61–78 (1990). https://doi.org/10.1145/91556.91592
199. Wadler, P.: The essence of functional programming. In: Principles of Programming Languages. ACM, pp. 1–14 (1992). https://doi.org/10.1145/143165.143169
200. Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015)
201. Wile, D.: POPART: producer of parsers and related tools: System builder's manual. USC/ISI Information Science Institute, University of Southern California, Technical report (1981)
202. Wile, D.: Program developments as formal objects. USC/ISI Information Science Institute, University of Southern California, Technical report (1981)
203. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Principles of Programming Languages, pp. 214–227 (1999)