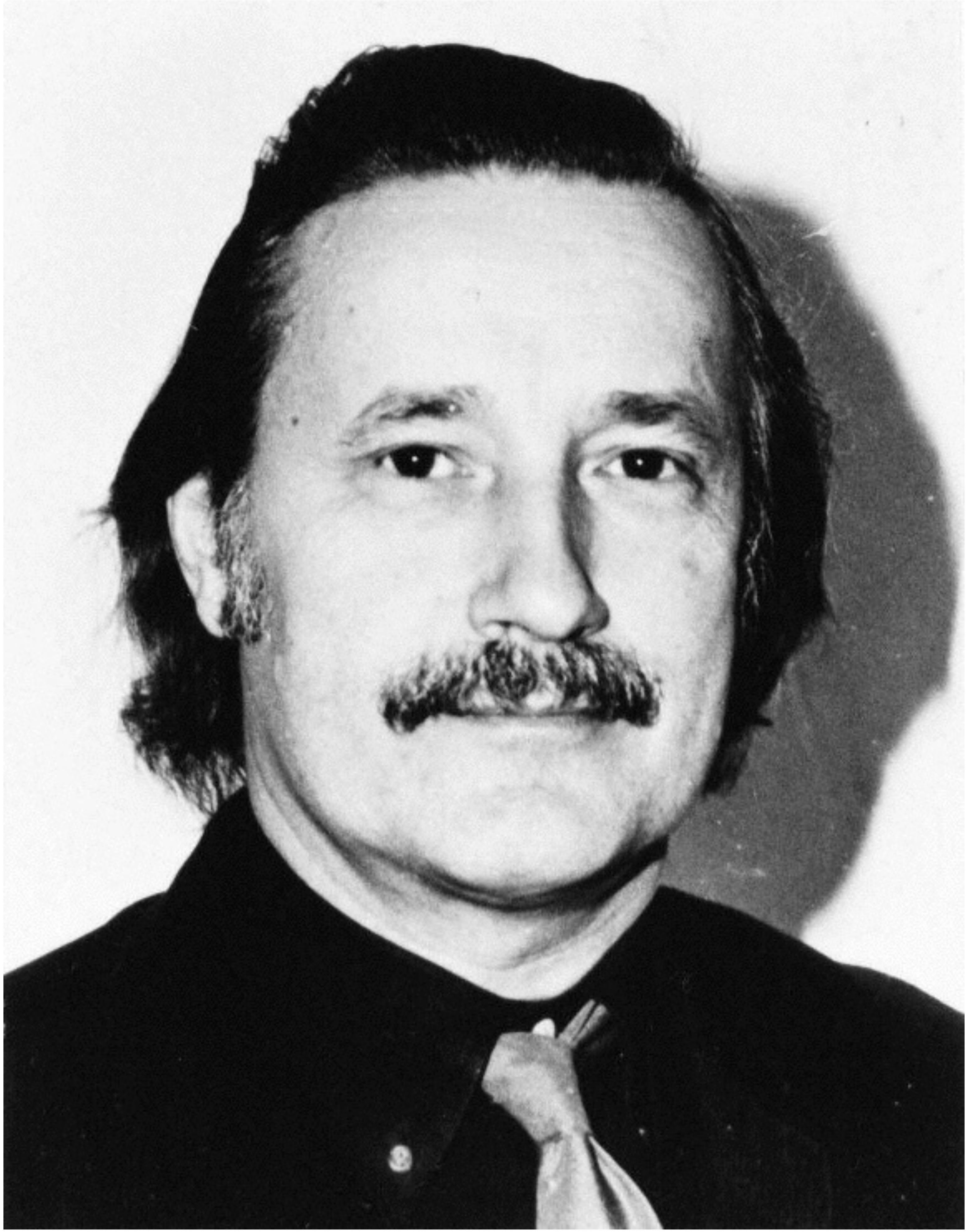
Christopher Strachey: First-class Citizen

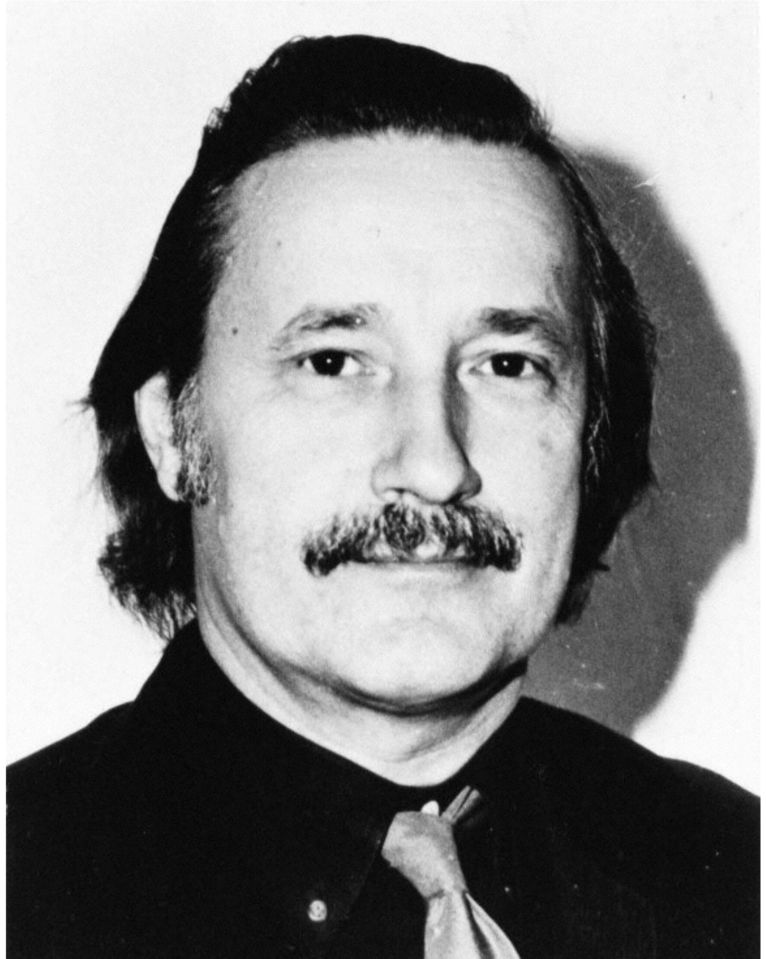
Philip Wadler
University of Edinburgh
Strachey 100, 19 November 2016

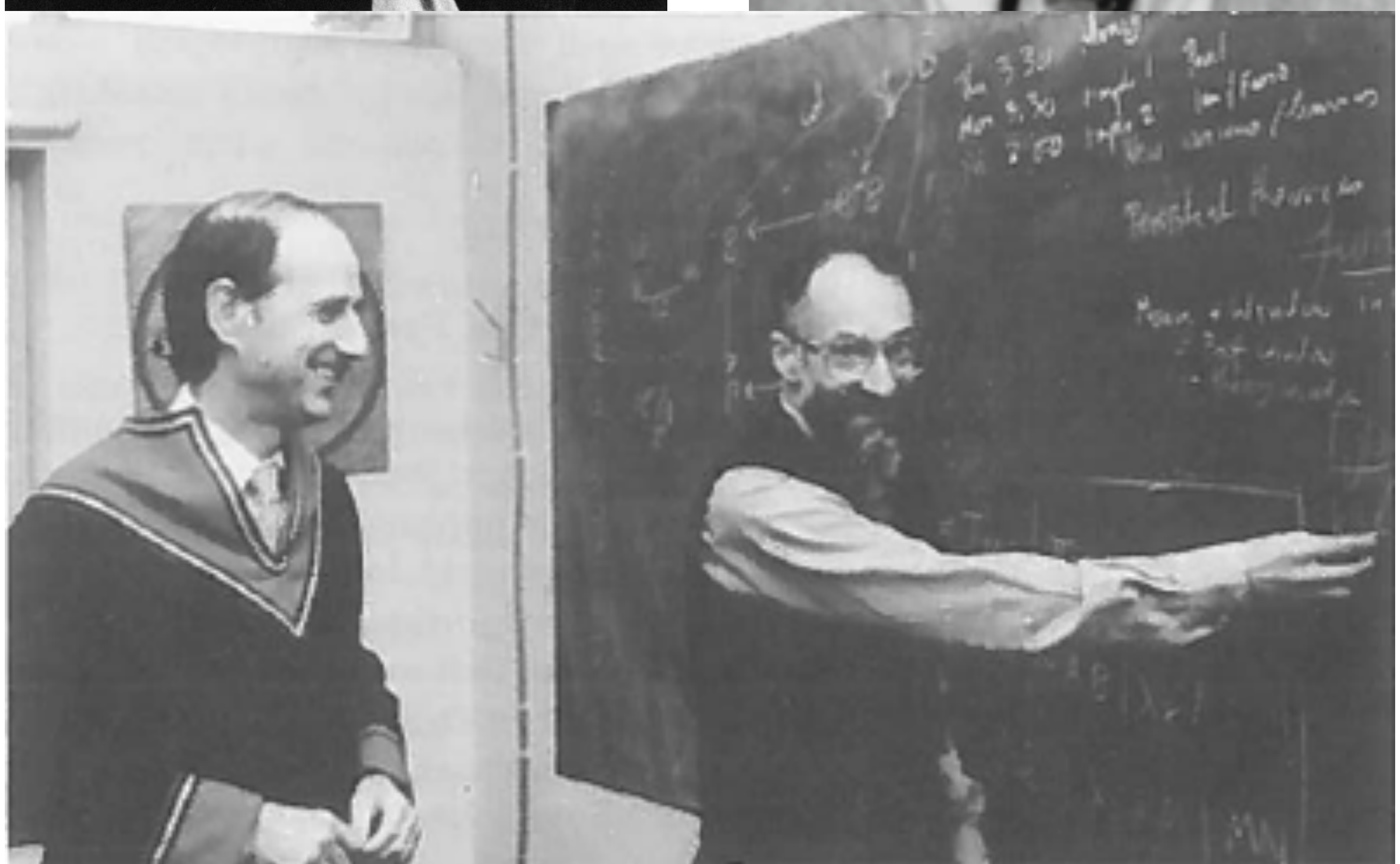


Mervyn Pragnell

M. Pragnell


FOYLES
WELCOME TO FOYLES
FOYLES
FOYLES





B is for Bonnie
C is for Christopher

CPL

Combined Programming Language
Cambridge Programming Language
Christopher's Programming Language

BCPL

BCPL

BCPL

SECOND EDITION

THE

PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

Fundamental Concepts in Programming Languages

Higher-Order and Symbolic Computation, 13, 11–49, 2000

Fundamental Concepts in Programming Languages

CHRISTOPHER STRACHEY
Reader in Computation at Oxford University, Programming Research Group, 45 Banbury Road, Oxford, UK

Abstract. This paper forms the substance of a course of lectures given at the International Summer School in Computer Programming at Copenhagen in August, 1967. The lectures were originally given from notes and the paper was written after the course was finished. In spite of this, and only partly because of the shortage of time, the paper still retains many of the shortcomings of a lecture course. The chief of these are an uncertainty of aim—it is never quite clear what sort of audience there will be for such lectures—and an associated switching from formal to informal modes of presentation which may well be less acceptable in print than it is natural in the lecture room. For these (and other) faults, I apologise to the reader.

STRACHEY - FUNDAMENTAL CONCEPTS

PLW HS ~~PDM~~

Int. Summer School in Computer Programming, Copenhagen, Aug. 1967 (not published)

Fundamental Concepts in Programming Languages

by

Christopher Strachey

(Reader in Computation at Oxford University)

<u>Note</u> This document is intended for publication. It is made available as a preprint on the understanding that references or extracts will not be published prior to the publication of the original without the consent of the author.

Programming Research Group,
45 Banbury Road,
Oxford.

PETER SESTOFT
1987-03-05
(FROM PETER D. MOSSES)

Contents

Functions as First-Class Citizens

3.5 Functions and Routines as Data Items

3.5.1 First and Second Class Objects

In Algol a real number may appear in an expression or be assigned to a variable, and either may appear as an actual parameter in a procedure call. A procedure, on the other hand, may only appear in another procedure call either as the operator (the most common case) or as one of the actual parameters. There are no other expressions involving procedures or whose results are procedures. Thus in a sense procedures in Algol are second class citizens - they always have to appear in person and can never be represented by a variable or expression (except in the case of a formal parameter), while we can write (in Algol still)

(<u>if</u> x > 1 <u>then</u> a <u>else</u> b) + 6

when a and b are reals, we cannot correctly write

(<u>if</u> x > 1 <u>then</u> Sin <u>else</u> Cos)(x)

nor can we write a type procedure (Algol's nearest approach to a function) with a result which is itself a procedure.

3.5. *Functions and routines as data items.*

3.5.1. First and second class objects. In ALGOL a real number may appear in an expression or be assigned to a variable, and either may appear as an actual parameter in a procedure call. A procedure, on the other hand, may only appear in another procedure call either as the operator (the most common case) or as one of the actual parameters. There are no other expressions involving procedures or whose results are procedures. Thus in a sense procedures in ALGOL are second class citizens—they always have to appear in person and can never be represented by a variable or expression (except in the case of a formal parameter), while we can write (in ALGOL still)

```
(if  x > 1 then a else b) + 6
```

when `a` and `b` are reals, we cannot correctly write

```
(if  x > 1 then sin else cos)(x)
```

nor can we write a type procedure (ALGOL's nearest approach to a function) with a result which is itself a procedure.

Suppose P is an operator (called by some a 'functional') which operates on functions. The result of applying P to a function $f(x)$ is often written $P[f(x)]$. What then does $P[f(x+1)]$ mean? There are two possible meanings (a) we form $g(x) = f(x+1)$ and the result is $P[g(x)]$ or (b) we form $h(x) = P[f(x)]$ and the result is $h(x+1)$. In many cases these are the same but not always. Let

$$P[f(x)] = \begin{cases} \dfrac{f(x) - f(0)}{x} & \text{for } x \neq 0 \\ f'(x) & \text{for } x = 0 \end{cases}$$

Then if $f(x) = x^2$

$$P[g(x)] = P[x^2 + 2x + 1] = x + 2$$

while

$$h(x) = P[f(x)] = x$$

so that $h(x+1) = x + 1$.

This sort of confusion is, of course, avoided by using λ-expressions or by treating functions as first class objects. Thus, for example, we should prefer to write $(P[f])[x]$ in place of $P[f(x)]$ above (or, using the association rule $P[f][x]$ or even $P\ f\ x$). The two alternatives which were confused would then become

$$P\ g\ x \quad \text{where } g\ x = f(x+1)$$

and $P\ f\ (x+1)$.

The first of these could also be written $P(\lambda x.\ f(x+1))x$.

I have spent some time on this discussion in spite of its apparently trivial nature, because I found, both from personal experience and from talking to others, that it is remarkably difficult to stop looking on functions as second class objects. This is particularly unfortunate as many of the more interesting developments of programming and programming languages come from the unrestricted use of functions, and in particular of functions which have functions as a result. As usual with new or unfamiliar ways of looking at things, it is harder for the teachers to change their habits of thought than it is for their pupils to follow them. The

I have spent some time on this discussion in spite of its apparently trivial nature, because I found, both from personal experience and from talking to others, that it is remarkably difficult to stop looking on functions as second class objects. This is particularly unfortunate as many of the more interesting developments of programming and programming languages come from the unrestricted use of functions, and in particular of functions which have functions as a result. As usual with new or unfamiliar ways of looking at things, it is harder for the teachers to change their habits of thought than it is for their pupils to follow them. The

Polymorphism

The desire to do this leads to an examination of the various forms of polymorphism. There seem to be two main classes, which can be called ad hoc polymorphism and parametric polymorphism.

In ad hoc polymorphism there is no single systematic way of determining the type of the result from the type of the arguments. There may be several rules of limited extent which reduce the number of cases, but these are themselves ad hoc both in scope and content. All the ordinary arithmetic operators and functions come into this category. It seems, moreover, that the automatic insertion of transfer functions by the compiling system is limited to this class.

Parametric polymorphism is more regular and may be illustrated by an example. Suppose `f` is a function whose argument is of type α and whose results is of β (so that the type of `f` might be written $\alpha \Rightarrow \beta$), and that `L` is a list whose elements are all of type α (so that the type of `L` is α **`list`**). We can imagine a function, say `Map`, which applies `f` in turn to each member of `L` and makes a list of the results. Thus `Map[f,L]` will produce a β **`list`**. We would like `Map` to work on all types of list provided f was a suitable function, so that `Map` would have to be polymorphic. However its polymorphism is of a particularly simple parametric type which could be written

$$(\alpha \Rightarrow \beta, \alpha\ \texttt{\textbf{list}}) \Rightarrow \beta\ \texttt{\textbf{list}}$$

where α and β stand for any types.

The desire to do this leads to an examination of the various forms of polymorphism. There seem to be two main classes, which can be called ad hoc polymorphism and parametric polymorphism.

In ad hoc polymorphism there is no single systematic way of determining the type of the result from the type of the arguments. There may be several rules of limited extent which reduce the number of cases, but these are themselves ad hoc both in scope and content. All the ordinary arithmetic operators and functions come into this category. It seems, moreover, that the automatic insertion of transfer functions by the compiling system is limited to this class.

Parametric polymorphism is more regular and may be illustrated by an example. Suppose `f` is a function whose argument is of type α and whose results is of β (so that the type of `f` might be written $\alpha \Rightarrow \beta$), and that `L` is a list whose elements are all of type α (so that the type of `L` is α **`list`**). We can imagine a function, say `Map`, which applies `f` in turn to each member of `L` and makes a list of the results. Thus `Map[f,L]` will produce a β **`list`**. We would like `Map` to work on all types of list provided f was a suitable function, so that `Map` would have to be polymorphic. However its polymorphism is of a particularly simple parametric type which could be written

$$(\alpha \Rightarrow \beta, \alpha\ \mathtt{\textbf{list}}) \Rightarrow \beta\ \mathtt{\textbf{list}}$$

where α and β stand for any types.

Polymorphism of both classes presents a considerable challenge to the language designer, but it is not one which we shall take up here.

Type Classes

How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal

ML [HMM86, Mil87], Miranda[1][Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

The type system presented here is a generalisation of the Hindley/Milner type system. As in that system, type declarations can be inferred, so explicit

1 Introduction

Strachey chose the adjectives *ad-hoc* and *parametric* to distinguish two varieties of *polymorphism* [Str67].

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in `3*3`) and multiplication of floating point values (as in `3.14*3.14`).

Parametric polymorphism occurs when a function is defined over a range of types, acting in the same way for each type. A typical example is the `length` function, which acts in the same way on a list of integers and a list of floating point numbers.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

Scala

MERCURY

Isabelle

C++

Type classes

Haskell
Clean
Mercury
Hal
Isabelle
Coq
Agda
Scala
C++ concepts
Rust

Semantics vs Syntax

This is probably an unfair criticism, for, as will become clear later, I am not only temperamentally a Platonist and prone to talking about abstracts if I think they throw light on a discussion, but I also regard syntactical problems as essentially irrelevant to programming languages at their present stage of development. In a rough and ready sort of way it seems to me fair to think of the semantics as being what we want to say and the syntax as how we have to say it. In these terms the urgent task in programming languages is to explore the field of semantic possibilities. When we have discovered the main outlines and the principal peaks we can set about devising a suitably neat and satisfactory notation for them, and this is the moment for syntactic questions.

This is probably an unfair criticism, for, as will become clear later, I am not only temperamentally a Platonist and prone to talking about abstracts if I think they throw light on a discussion, but I also regard syntactical problems as essentially irrelevant to programming languages at their present stage of development. In a rough and ready sort of way it seems to me fair to think of the semantics as being what we want to say and the syntax as how we have to say it. In these terms the urgent task in programming languages is to explore the field of semantic possibilities. When we have discovered the main outlines and the principal peaks we can set about devising a suitably neat and satisfactory notation for them, and this is the moment for syntactic questions.