



Part I

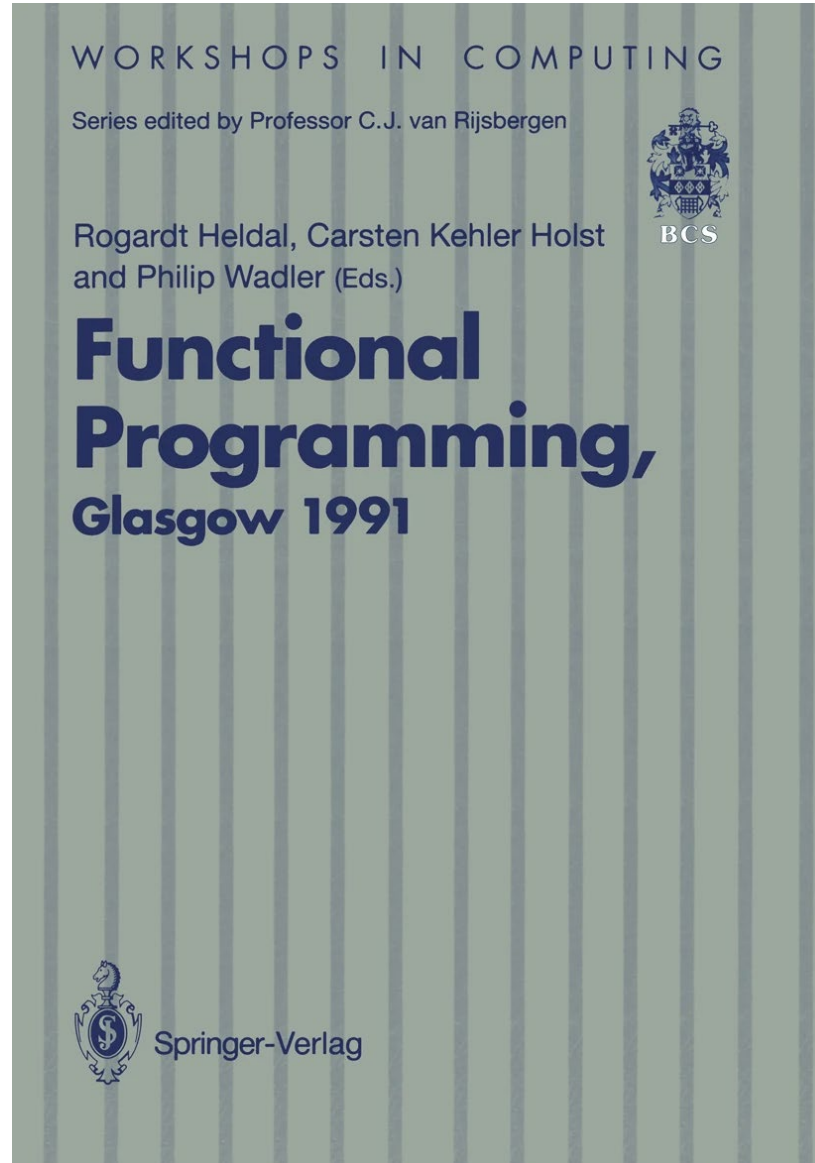
John Hughes

● Glasgow

● Chalmers



1988: Rothesay
1989: Fraserburgh
1990: Ullapool
1991: Portree
1992: Ayr
1993: Ayr
1994: Ayr
1995: Ullapool
1996: Ullapool
1998: Pitlochry



1999: Scottish Functional
Programming Workshop
2003: Trends in Functional
Programming





A Lazy Evaluator

Peter Henderson
University of Newcastle upon Tyne

James H. Morris, Jr.
Xerox Palo Alto Research Center

to execute pure LISP programs is
valuation of parameters and list

Thus a lazy evaluator
second parameter of

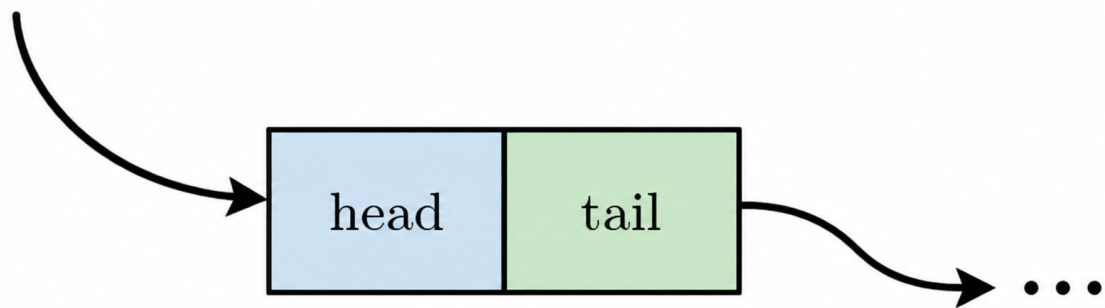
D.P. Friedman
D.S. Wise

CONS SHOULD NOT EVALUATE ITS ARGUMENTS

The *cons* constructor function which allocates and
records in recursive, side-effect-free proce-
languages is redefined to be a *non-strict* (Friedman
min 1974) elementary operation. Instead of
evaluating its arguments, it builds suspensions
which are not coerced until the suspension is
accessed by a strict elementary function. The
resulting evaluation procedures are strict.



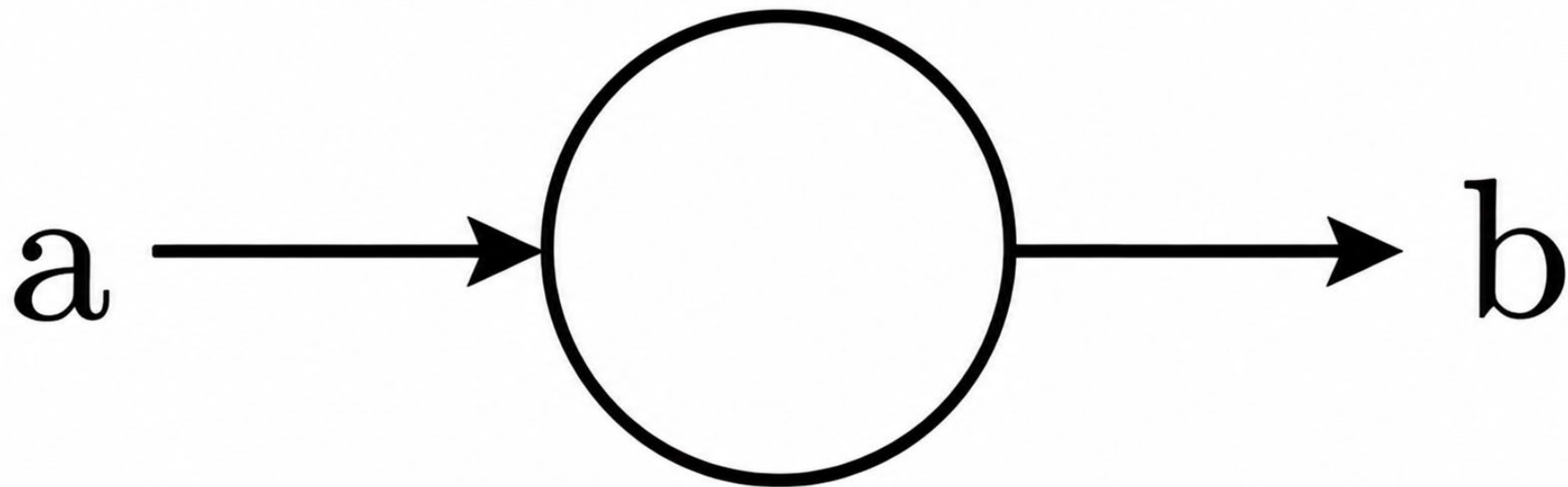
1976 SASL
1981 Kent Recursive Calculator
1983 Miranda



A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
-------	-------	-------	-------	-------	-------	-------	-------

var

$[a] \rightarrow [b]$



head map filter
init (++) tail
last concat
take drop takeWhile
dropWhile foldr foldl
scan

+

list comprehensions

1988

Richard Bird
Philip Wadler
**Introduction to
Functional
Programming**

PRENTICE HALL
INTERNATIONAL
SERIES IN
COMPUTER
SCIENCE

C.A.R. HOARE SERIES EDITOR

Why Functional Programming Matters

John Hughes, Institutionen för Datavetenskap,
Chalmers Tekniska Högskola,
41296 Göteborg,
SWEDEN. rjmh@cs.chalmers.se

This paper dates from 1984, and circulated as a Chalmers memo for many years. Slightly revised versions appeared in 1989 and 1990 as [Hug90] and [Hug89]. This version is based on the original Chalmers memo `nroff` source, lightly edited for LaTeX and to bring it closer to the published versions, and with one or two errors corrected. Please excuse the slightly old-fashioned type-setting, and the fact that the examples are not in Haskell!

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write,

How to Replace Failure by a List of Successes
A method for exception handling, backtracking, and pattern matching
in lazy functional languages

Philip Wadler
Programming Research Group, Oxford University
11 Keble Road, Oxford, OX1 3QD

Abstract

Should special features for exception handling, backtracking, or pattern matching be included in a programming language? This paper presents a method whereby some programs that use these features can be re-written in a functional language with lazy evaluation, without the use of any special features. This method may be of use for practicing functional programmers; in addition, it provides further evidence of the power of lazy evaluation. The method itself is straightforward: each term that may raise an exception or backtrack is replaced by a term that returns a list of values. In the special case of pattern matching without backtracking, the method can be applied



reverse [1..]

Listlessness is Better than Laziness

Lazy evaluation and garbage collection at compile-time

Philip Wadler
Programming Research Group, Oxford University
8-11 Keble Rd., Oxford, OX1 3QD

0. Introduction

One reason for interest in functional programming is that it is well-suited for program development by transformation: begin by writing a clear (but inefficient) program, and transform this to an efficient (but less clear) program. Promising research has occurred in this area [Burstall and Darlington 77; Clark and Darlington 80; Feather 82; Manna and Waldinger 79; Scherlis 81] and it is ripe for further exploration.

Most desirable is a transformation system that can always automatically transform a program to optimal form. It is unreasonable to expect this in general, but it may be possible over limited domains.

This paper is mainly a summary of the author's Ph.D. thesis [Wadler 84], but it also includes a few results that are not in the thesis.

This paper is organized as follows. Section 1 explains why intermediate lists are a source of both clarity and inefficiency. Section 2 gives an example of the transformation method. Section 3 presents the machine for executing transformed programs. Section 4 outlines theoretical results and extensions to the method. Section 5 gives a more realistic example of program transformation, using a text processing method inspired by Unix. Section 6 discusses related work. Section 7 suggests topics for future work and presents conclusions.

Theoretical Computer Science 73 (1990) 231–248
North-Holland

231

DEFORESTATION: TRANSFORMING PROGRAMS TO ELIMINATE TREES *

Philip WADLER

Department of Computer Science, University of Glasgow, Glasgow G12 8QQ, UK

Abstract. An algorithm that transforms programs to eliminate intermediate trees is presented. The algorithm applies to any term containing only functions with definitions in a given syntactic form, and is suitable for incorporation in an optimizing compiler.

0. Introduction

A Short Cut to Deforestation

Andrew Gill John Launchbury Simon L Peyton Jones
 Department of Computing Science, University of Glasgow G12 8QQ
 {andy,jl,simonpj}@dcs.glasgow.ac.uk

Abstract

Lists are often used as “glue” to connect separate parts of a program together. We propose an automatic technique for improving the efficiency of such programs, by removing many of these intermediate lists, based on a single, simple, local transformation. We have implemented the method in the Glasgow Haskell compiler.

1 Introduction

Functional programs are often constructed by combining to-

```
all' p xs = h xs
  where h []      = True
        h (x:xs) = p x && h xs
```

Now no intermediate list is used, but this has been achieved at the cost of clarity and conciseness compared with the original definition.

We want to eat our cake and have it too. That is, we would like to write our programs in the style of `all`, but have the compiler automatically transform this into the more efficient version `all'`.

One example of just such a transformation is *deforestation* (Wells, 1986). Deforestation is a technique for removing

Theorems for free!

Philip Wadler
University of Glasgow*

Abstract

From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

1 Introduction

Write down the definition of a polymorphic function on

rearrange such lists, independent of the values contained in them. Thus applying a to each element of a list and then rearranging yields the same result as rearranging and then applying a to each element.

For instance, r may be the function *reverse* : $\forall X. X^* \rightarrow X^*$ that reverses a list, and a may be the function *code* : *Char* $\rightarrow$ *Int* that converts a character to its ASCII code. Then we have

$$\begin{aligned} & \text{code}^* (\text{reverse}_{\text{Char}} ['a', 'b', 'c']) \\ &= [99, 98, 97] \\ &= \text{reverse}_{\text{Int}} (\text{code}^* ['a', 'b', 'c']) \end{aligned}$$



Report on the Programming Language Haskell

A Non-strict, Purely Functional Language

Version 1.0

1 April 1990

Paul Hudak¹ [editor]
Philip Wadler² [editor]
Andrew Chalkley³

The type of equality: (==)

Miranda: $* \rightarrow * \rightarrow \text{Bool}$

Standard ML: $'a \rightarrow 'a \rightarrow \text{Bool}$

Arithmetic

Miranda

Standard ML

CaML

2+2

num

int

int

2.0 + 2.0

num

real

2.0 +. 2.0

2 + 2.5

num

X

X

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 introduction to type classes, and defines them for-

ML [HMM86, Mil87], Miranda¹[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 type declarations for functions are not required. Dur-

```
class Eq a where  
    (==) :: a -> a -> Bool
```

```
instance Eq Int where  
    x == y = ...
```

```
elem :: Eq a => a -> [a] -> Bool
```

```
class Num a where
```

```
    (+) :: a -> a -> a
```

```
    (-) :: a -> a -> a
```

```
    (*) :: a -> a -> a
```

```
    ...
```

```
instance Num Int where...
```

```
instance Num Float where...
```

```
instance Num (Complex Float) where...
```

```
double :: Num a => a -> a
```

```
double x = x + x
```

```
instance Num a where
```

```
...
```

```
fromInteger :: Integer -> a
```

2 + 2.5

fromInteger

MacQueen: type classes SML modules

class

signature

instance

structure

overloaded

functor

function

Haskell 1.0: 12 classes

Constructor classes

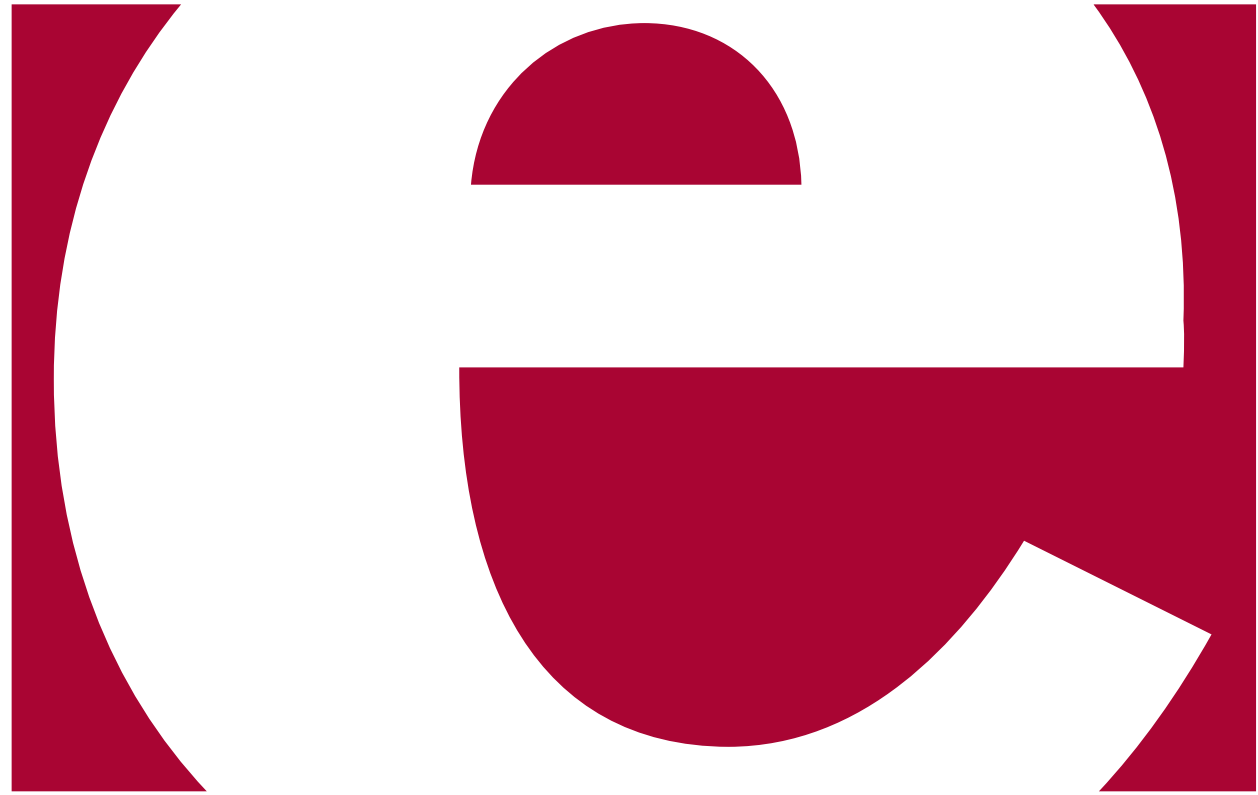
Multi-parameter type classes

Functional dependencies

Associated types + data-types

Monad, Functor, Applicative

Monad transformer libraries



ERLANG

Functional+ concurrency and distribution

Designed at



in Stockholm

Brought functional programming to telecoms industry,
internet servers (e.g. WhatsApp)

Strict ☹️

Dynamically typed ☹️☹️☹️

1997

A Practical Subtyping System For Erlang

Simon Marlow

`simonm@dcs.gla.ac.uk`

University of Glasgow

Philip Wadler

`wadler@research.bell-labs.com`

Bell Labs, Lucent Technologies

Type specs
written in
comments

Dialyzer

“Who needs
soundness
anyway?”

Gradual
typing

e.g.
WhatsApp
eqWAlizer

1 Introduction

We can stop waiting for functional languages to be used in

variables, the basis of t, Fagan, and low variable s natural to us, to construc truck us as h pping systems removing run information is ng. alternative that we based on subtyping. Type systems with subtyping have been studied by several researchers [Mit91, FM88, MR85, Rey85], and are based on solving systems of typing constraints of the form $U \subset V$, where U and V are types. Hindley-Milner

2007





● Chalmers

● Oxford



...to be continued













