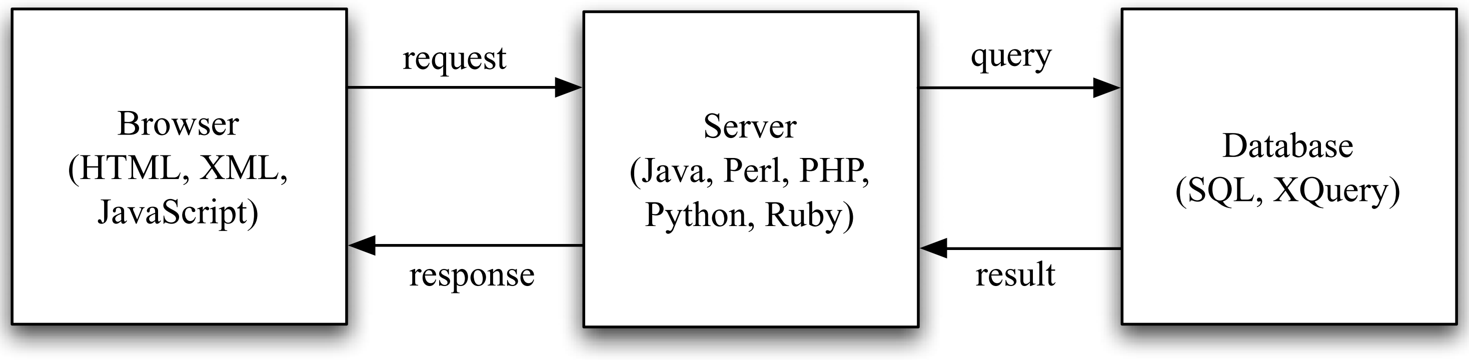
Formlets

Philip Wadler
University of Edinburgh
`wadler@inf.ed.ac.uk`

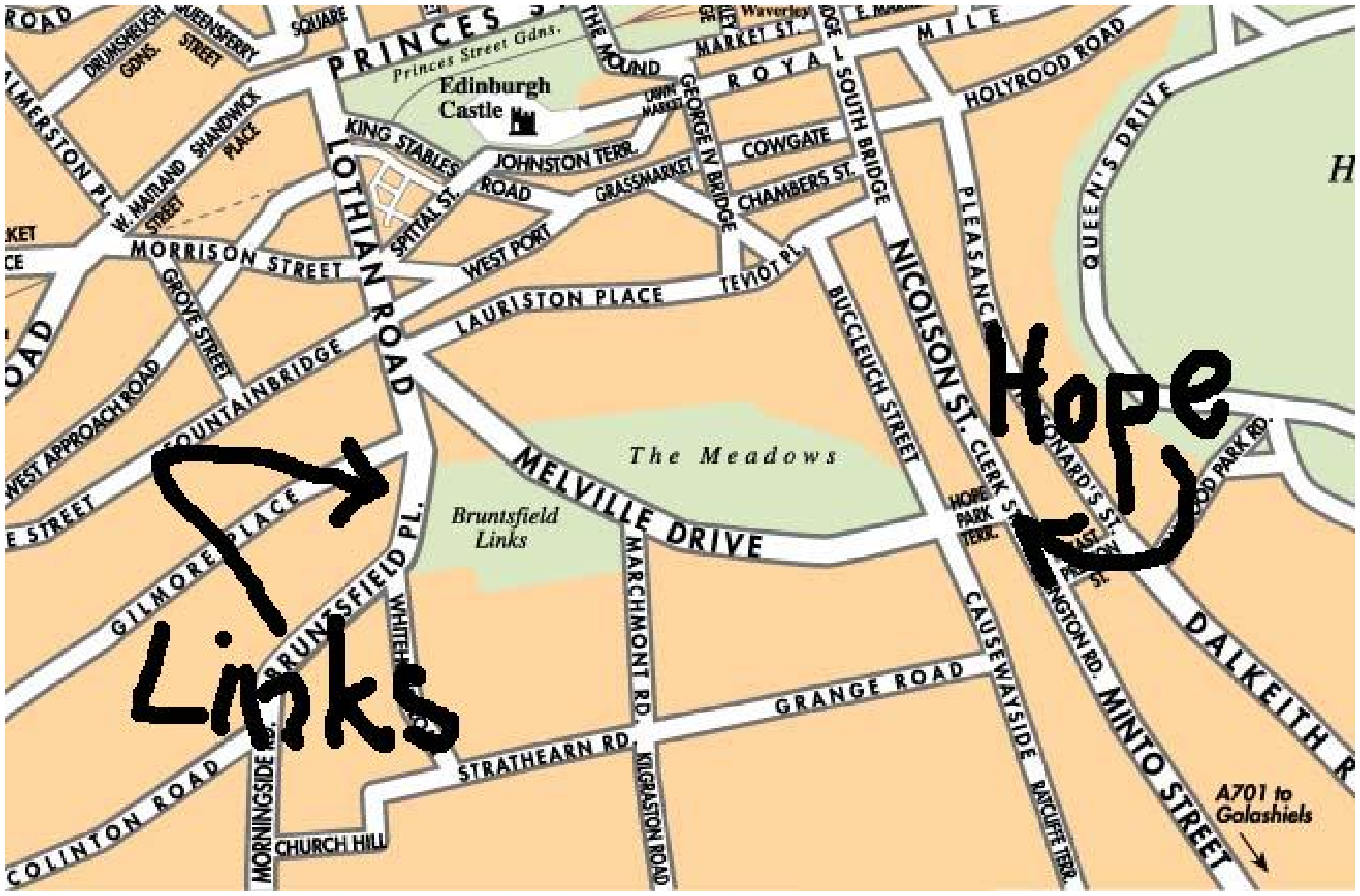
The Adventure of the Absent Abstraction

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

Three Tiers


Browser
(HTML, XML, JavaScript)
request
response
Server
(Java, Perl, PHP, Python, Ruby)
query
result
Database
(SQL, XQuery)

Links: Web Programming without Tiers



Links, before formlets

```
let request : Xml =
  <form l:action="{
    let start  = make_date(string_to_int(sm),
                           string_to_int(sd)) in
    let finish = make_date(string_to_int(fm),
                           string_to_int(fd)) in
    response(start,finish)
  }">
    Start:  month <input l:name="{sm}"/>
            day   <input l:name="{sd}"/>
    Finish: month <input l:name="{fm}"/>
            day   <input l:name="{fd}"/>
    <input type="submit" value="Submit"/>
  </form>
```

iData For The World Wide Web
Programming Interconnected Web Forms

Rinus Plasmeijer and Peter Achten

Software Technology, Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, Toernooiveld 1, 6525ED Nijmegen, Netherlands

```
counterIData :: IDataId Int → IDataFun Int
counterIData iDataId i           = mkIData iDataId i ibm
where ibm        = { toView      = λn v → useOldView (n,down,up) v
                   , updView     = λ_ v → updCounter v
                   , fromView    = λ_ (n,_,_) → n
                   , resetView   = Nothing }
      (up,down) = (LButton (defpixel / 6) "+",LButton (defpixel / 6) "-")

      updCounter :: Counter → Counter
      updCounter (n,Pressed,_) = (n - 1,down,up)
      updCounter (n,_,Pressed) = (n + 1,down,up)
      updCounter noPresses     = noPresses

      useOldView new (Just old)= old
      useOldView new Nothing   = new
```

Formlets

The Essence of Form Abstraction*

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop

School of Informatics, University of Edinburgh

```
module type Idiom = sig
  type α t
  val pure  : α → α t
  val (⊗) : (α → β) t → α t → β t
end
```

```
module type FORMLET = sig
  include Idiom
  val xml   : xml → unit t
  val text  : string → unit t
  val tag   : tag → attrs → α t → α t
  val input : string t
  val run   : α t → xml × (env → α)
end
```

Fig. 4. The idiom and formlet interfaces

Links, with formlets, sugared

```
let date_formlet : Formlet Date =
  formlet
    month {int_formlet => month}
    day   {int_formlet => day}
  yields (make_date(month,day))

let dates_formlet : Formlet (Date,Date) =
  formlet
    Start:  {date_formlet => start}
    Finish: {date_formlet => finish}
    <input type="submit" value="submit"/>
  yields (start,finish)

let request : Xml =
  handle(dates_formlet, response)
```

Links, with formlets, unsugared

```
let int_formlet : Formlet Int =
  pure(string_to_int) @ input

let date_formlet : Formlet Date =
  pure(fun month day => make_date(month,day))
  @ text("month ") @ int_formlet
  @ text("day   ") @ int_formlet

let dates_formlet : Formlet (Date,Date) =
  pure(fun () start () finish () => (start,finish))
  @ text("Start:  ") @ date_formlet
  @ text("Finish: ") @ date_formlet
  @ submit("Submit")

let request : Xml = handle(dates_formlet,response)
```

Currying

$$(a, b) \to c \;\simeq\; a \to (b \to c)$$

```
      (fun (x,y) => x+y) (3,4)
⟹    3+4
⟹    7
```

```
      (fun x y => x+y) 3 4
≡     ((fun x => (fun y => x+y)) 3) 4
⟹    (fun y => 3+y) 4
⟹    3+4
⟹    7
```

Formlets in Links, the API

Formlets API

```
pure    : a -> Formlet a
(@)     : Formlet (a -> b) -> Formlet a -> Formlet b
text    : String -> Formlet ()
tag     : (Tag, Attrs, Formlet a) -> Formlet a
input   : Formlet String
run     : Formlet a -> (Xml, (Env -> a))
```

XML API

```
(^^)       : Xml -> Xml -> Xml
textXml    : String -> Xml
tagXml     : (Tag, Attrs, Xml) -> Xml
```

Formlets in Links, the implementation

```
type Formlet a = Int -> (Xml, (Env -> a), Int)

let pure(x)(i) = ([], (fun env => x), i)
let (f @ g)(i) =
  let (u, d, j) = f(i) in
  let (v, e, k) = g(j) in
  (u ^^ v, (fun env => d(env)(e(env))), k)
let xml(u)(i) = (u, (fun env => ()), i)
let text(s) = xml(xmlText(s))
let tag(t,a,f)(i) =
  let (u, d, j) = f(i) in
  (xmlTag(t,a,u), d, j)
let input(i) =
  let w = "i_" ^ string_of_int(i) in
  let u = xmlTag "input" [("name", w)] [] in
  (u, (fun env => lookUp(env,w)), i+1)
let run(f) =
  let (x, d, j) = f 0 in
  (x, d)
```

Monads

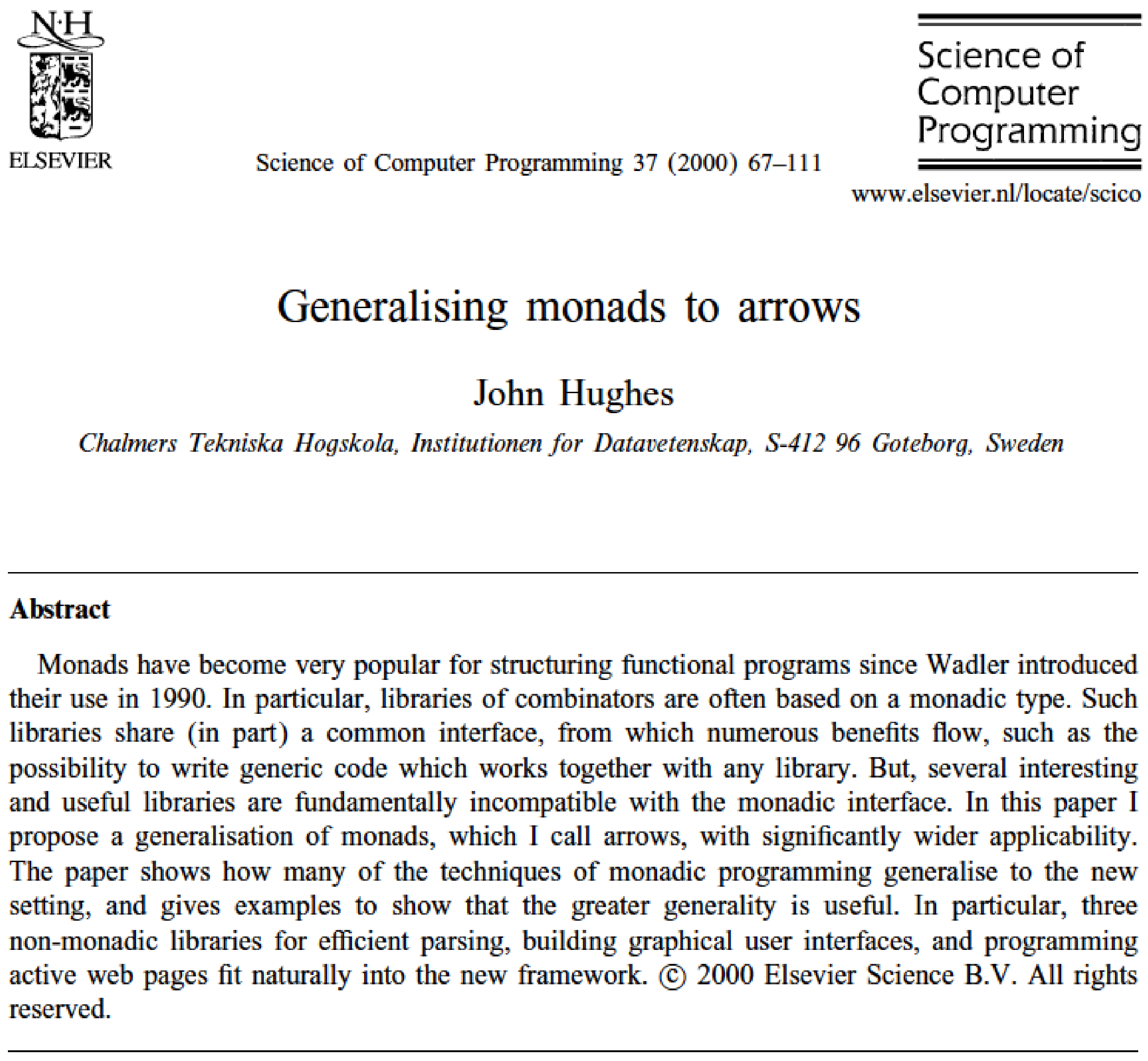
Notions of Computation and Monads

EUGENIO MOGGI*

Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, UK

$$\begin{tikzcd}
T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"'] & TA
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, "\eta_{TA}"] \arrow[dr, "\mathrm{id}_{TA}"'] & T^2A \arrow[d, "\mu_A"] & TA \arrow[l, "T\eta_A"'] \arrow[dl, "\mathrm{id}_{TA}"] \\
& TA &
\end{tikzcd}$$

Arrows



ELSEVIER

Science of Computer Programming 37 (2000) 67–111

Science of Computer Programming

www.elsevier.nl/locate/scico

Generalising monads to arrows

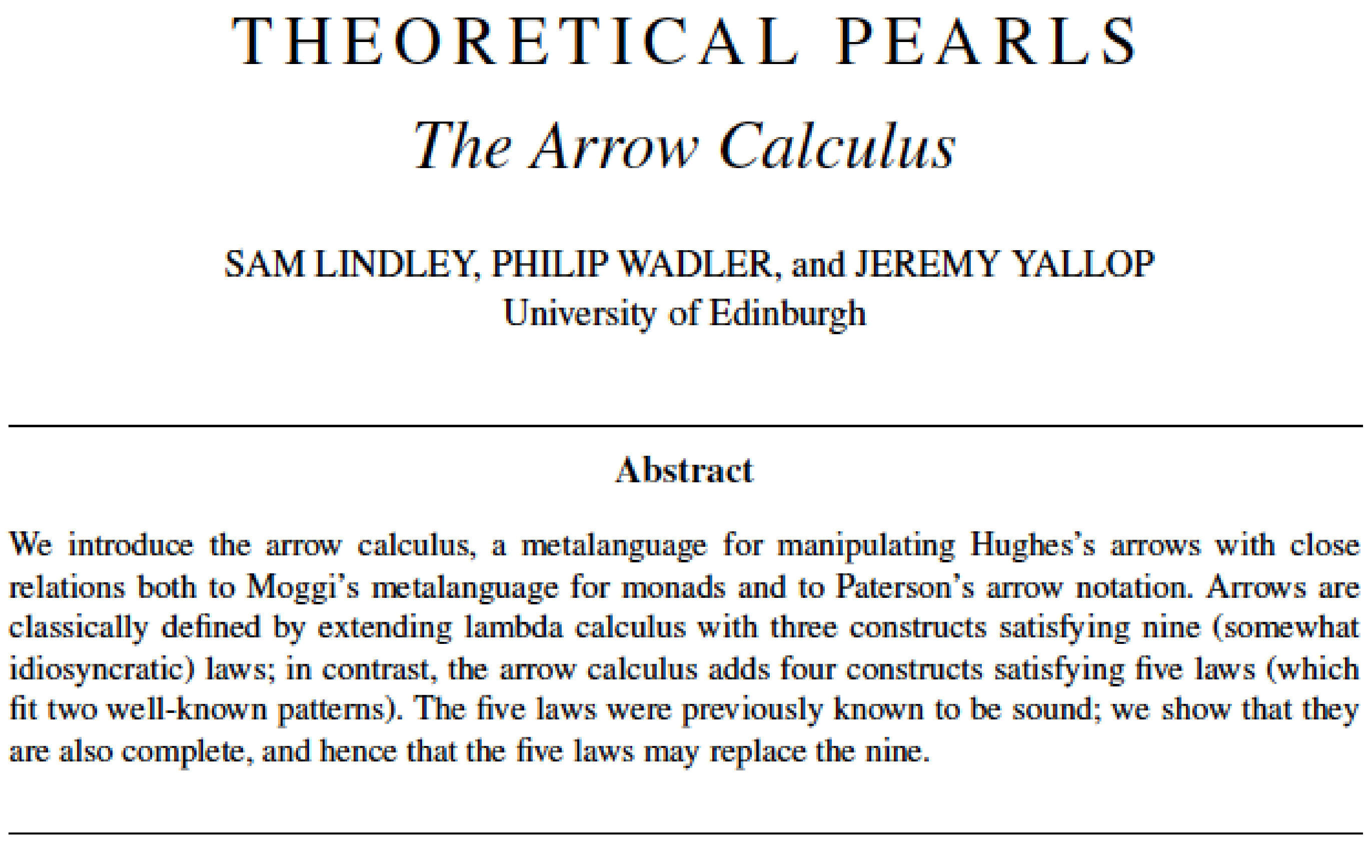
John Hughes

Chalmers Tekniska Hogskola, Institutionen for Datavetenskap, S-412 96 Goteborg, Sweden

Abstract

Monads have become very popular for structuring functional programs since Wadler introduced their use in 1990. In particular, libraries of combinators are often based on a monadic type. Such libraries share (in part) a common interface, from which numerous benefits flow, such as the possibility to write generic code which works together with any library. But, several interesting and useful libraries are fundamentally incompatible with the monadic interface. In this paper I propose a generalisation of monads, which I call arrows, with significantly wider applicability. The paper shows how many of the techniques of monadic programming generalise to the new setting, and gives examples to show that the greater generality is useful. In particular, three non-monadic libraries for efficient parsing, building graphical user interfaces, and programming active web pages fit naturally into the new framework. © 2000 Elsevier Science B.V. All rights reserved.

Arrow calculus



THEORETICAL PEARLS

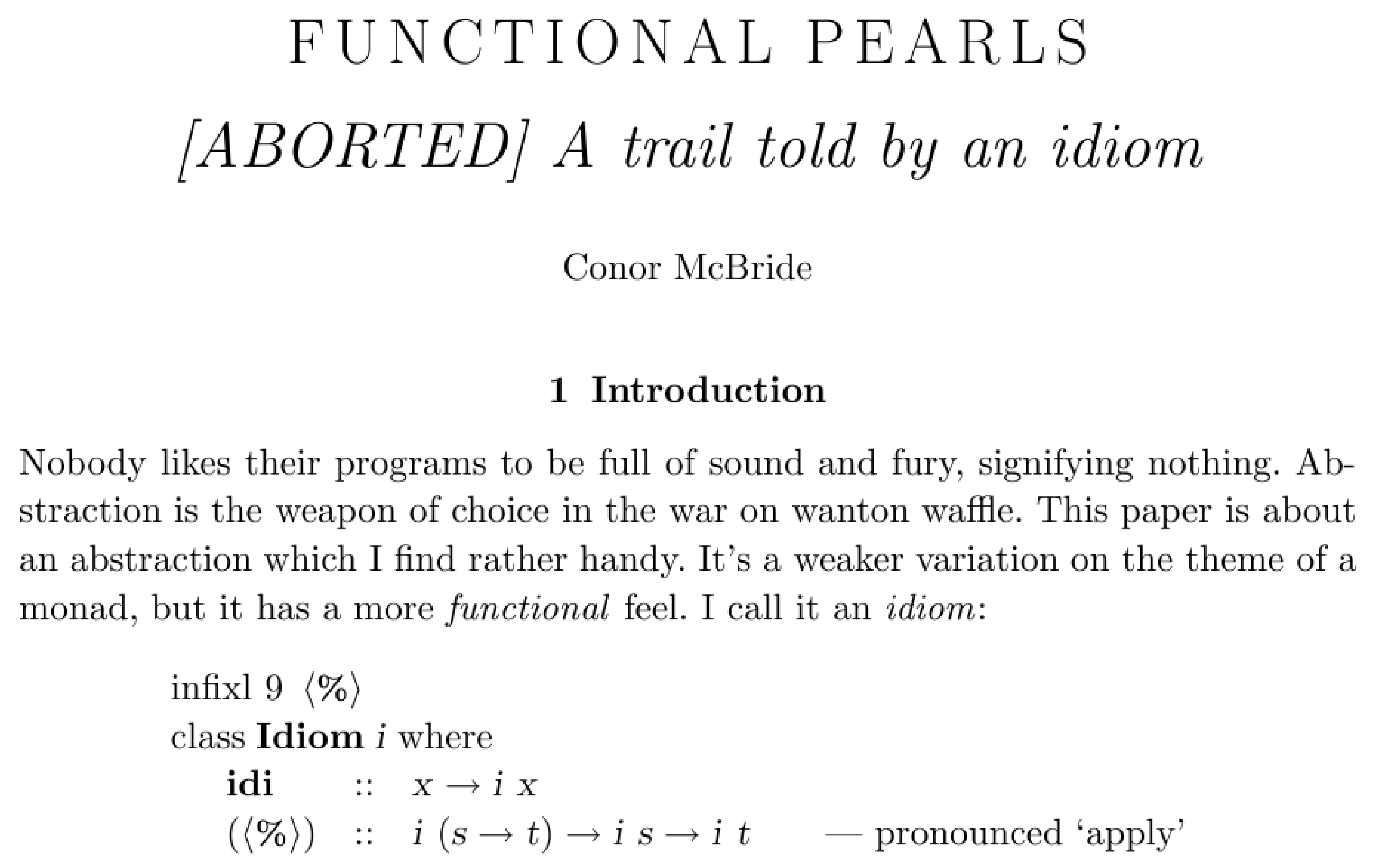
The Arrow Calculus

SAM LINDLEY, PHILIP WADLER, and JEREMY YALLOP
University of Edinburgh

Abstract

We introduce the arrow calculus, a metalanguage for manipulating Hughes's arrows with close relations both to Moggi's metalanguage for monads and to Paterson's arrow notation. Arrows are classically defined by extending lambda calculus with three constructs satisfying nine (somewhat idiosyncratic) laws; in contrast, the arrow calculus adds four constructs satisfying five laws (which fit two well-known patterns). The five laws were previously known to be sound; we show that they are also complete, and hence that the five laws may replace the nine.

Idioms



FUNCTIONAL PEARLS

[ABORTED] A trail told by an idiom

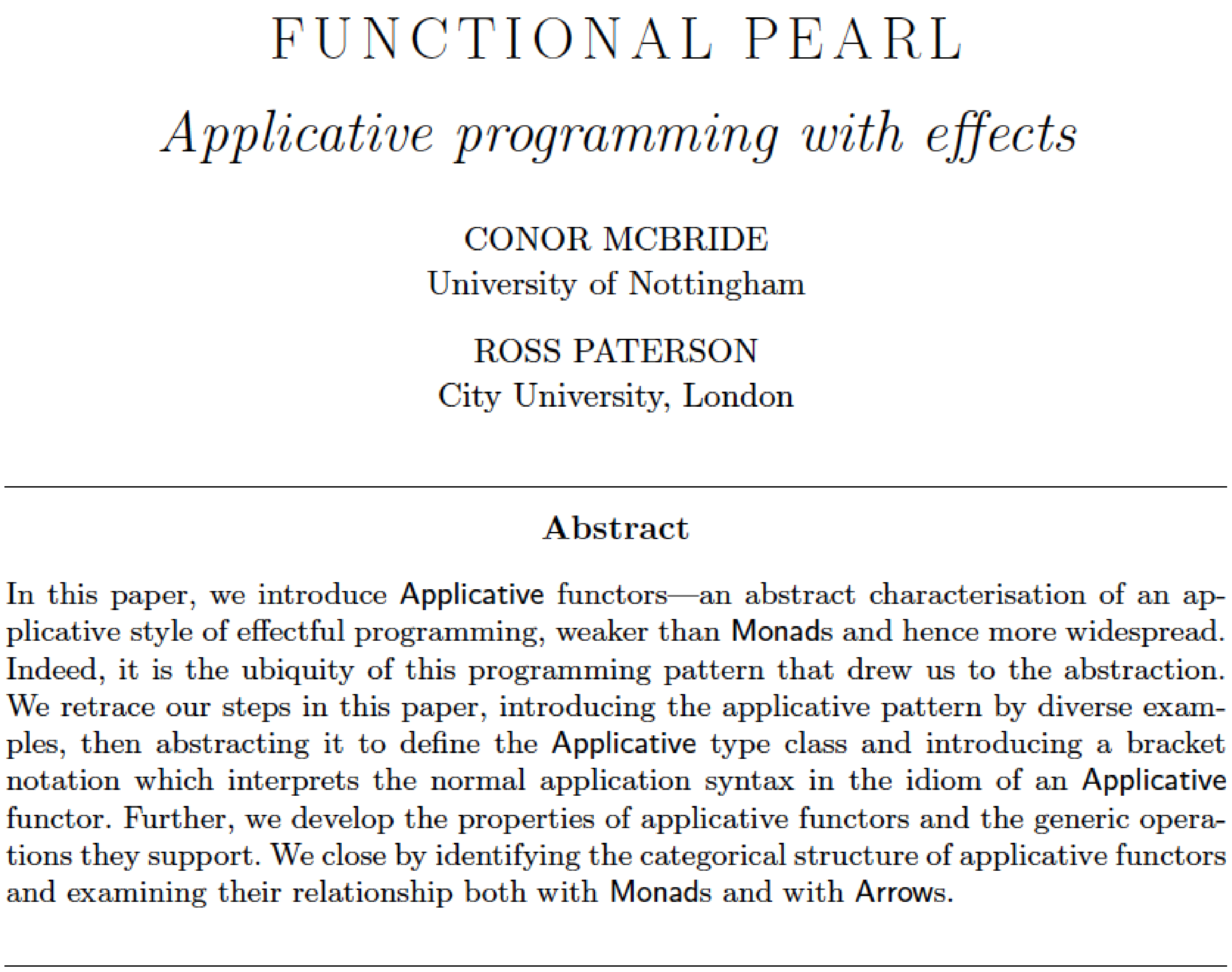
Conor McBride

1 Introduction

Nobody likes their programs to be full of sound and fury, signifying nothing. Abstraction is the weapon of choice in the war on wanton waffle. This paper is about an abstraction which I find rather handy. It's a weaker variation on the theme of a monad, but it has a more *functional* feel. I call it an *idiom*:

```
infixl 9 ⟨%⟩
class Idiom i where
    idi     ::  x → i x
    (⟨%⟩)   ::  i (s → t) → i s → i t       — pronounced ‘apply’
```

Idioms ⇒ Applicative Functors



FUNCTIONAL PEARL

Applicative programming with effects

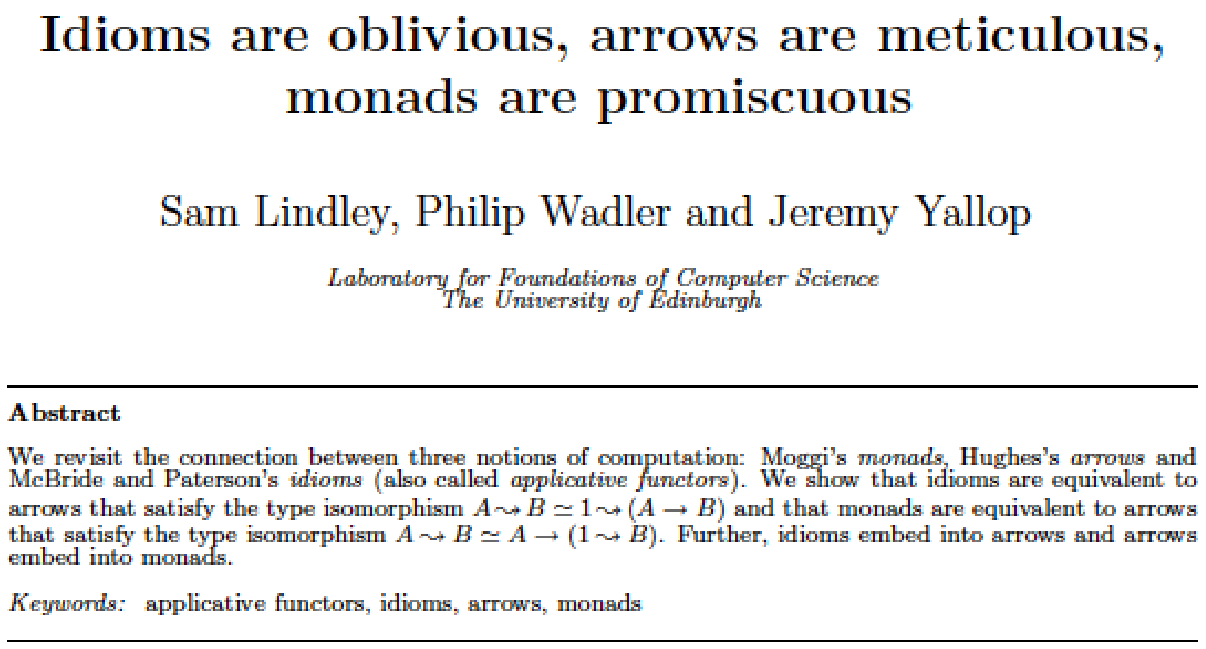
CONOR MCBRIDE
University of Nottingham

ROSS PATERSON
City University, London

Abstract

In this paper, we introduce Applicative functors—an abstract characterisation of an applicative style of effectful programming, weaker than Monads and hence more widespread. Indeed, it is the ubiquity of this programming pattern that drew us to the abstraction. We retrace our steps in this paper, introducing the applicative pattern by diverse examples, then abstracting it to define the Applicative type class and introducing a bracket notation which interprets the normal application syntax in the idiom of an Applicative functor. Further, we develop the properties of applicative functors and the generic operations they support. We close by identifying the categorical structure of applicative functors and examining their relationship both with Monads and with Arrows.

Monads < Arrows < Idioms



Idioms are oblivious, arrows are meticulous, monads are promiscuous

Sam Lindley, Philip Wadler and Jeremy Yallop

Laboratory for Foundations of Computer Science
The University of Edinburgh

Abstract

We revisit the connection between three notions of computation: Moggi's *monads*, Hughes's *arrows* and McBride and Paterson's *idioms* (also called *applicative functors*). We show that idioms are equivalent to arrows that satisfy the type isomorphism $A \rightsquigarrow B \simeq 1 \rightsquigarrow (A \to B)$ and that monads are equivalent to arrows that satisfy the type isomorphism $A \rightsquigarrow B \simeq A \to (1 \rightsquigarrow B)$. Further, idioms embed into arrows and arrows embed into monads.

Keywords: applicative functors, idioms, arrows, monads

$$\begin{tikzcd}
\text{idioms} \arrow[d, bend left] & & \text{monads} \arrow[d, bend left] \\
\text{static arrows} \arrow[u, bend left] \arrow[r, hook] & \text{arrows} \arrow[r, hook] & \text{higher-order arrows} \arrow[u, bend left] \\
(A \rightsquigarrow B \simeq 1 \rightsquigarrow (A \to B)) & & (A \rightsquigarrow B \simeq A \to (1 \rightsquigarrow B))
\end{tikzcd}$$

Idioms

```
pure : a -> Idiom a
(@)  : Idiom (a -> b) -> Idiom a -> Idiom b

pure (fun x => x) @ u
  =  u
pure (fun f g x => f (g x)) @ u @ v @ w
  =  u @ (v @ w)
pure f @ pure x
  =  pure (f x)
u @ pure x
  =  pure (fun f => f x) @ u
```

Programming the web

The IntelliFactory WebSharper™ Platform

Writing good web applications is not an easy task today. It requires a mastery of numerous languages (JavaScript, HTML, CSS), and an acute awareness of existing standards and browser implementation quirks. Poor debugging tools, and the lack of compositionality and component reuse in the multi-tiered, multi-language web environment compound the problem even more.

Seamless ASP.NET Integration

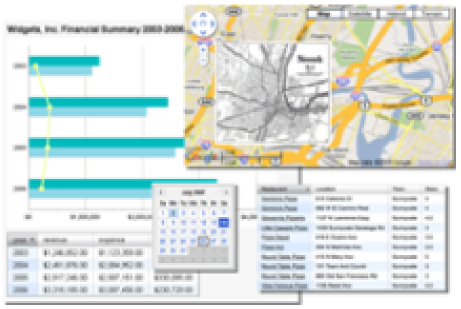
Plug your WebSharper™ applications into existing ASP.NET sites and deploy via IIS!

Functional Reactive Coding

Use powerful F# asynchronous constructs and first-class events with your client applications!

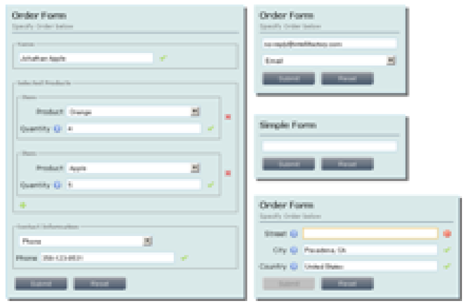
Extensions

Develop applications that use any JavaScript-based technology via WebSharper™ bindings!



Formlets

Create interactive forms with validation using type-safe code in just lines!



Implemented in O'Caml, Haskell, F#, Scheme.

Used by Tupil, Utrecht and IntelliFactory, Budapest

Are functional languages about to go mainstream?

Haskell, O'Caml, Racket

Erlang, Scala, F#