Haskell



Prof. dr. ir. Tom Schrijvers
tom.schrijvers@kuleuven.be

FP History

1936
λ calculus



Alonzo
Church

FP History

Functional Languages

1936
λ calculus



Alonzo
Church

FP History

Functional Languages

1936
λ calculus



Alonzo Church

1958
Lisp



John McCarthy

FP History

Functional Languages

1936
λ calculus



Alonzo Church

1958
Lisp



John McCarthy

1973
ML



Robin Milner

FP History

Functional Languages

1936
λ calculus



Alonzo Church

1958
Lisp



John McCarthy

1973
ML



Robin Milner

1987
Haskell



Haskell Committee

FP History

Functional Languages

Mainstream

1936
λ calculus



Alonzo Church

1958
Lisp



John McCarthy

1973
ML



Robin Milner

1987
Haskell



Haskell Committee

FP History

Functional Languages

Mainstream

1936
λ calculus



Alonzo Church

1958
Lisp



John McCarthy

1973
ML



Robin Milner

1987
Haskell



Haskell Committee

2007
C#

Microsoft

FP History

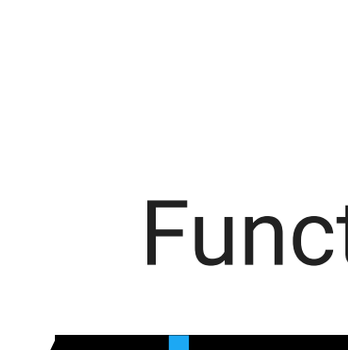
Functional Languages

Mainstream

1936
λ calculus

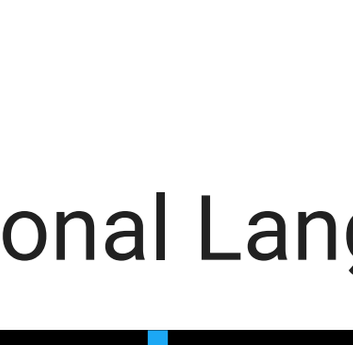
Alonzo Church

1958
Lisp



John McCarthy

1973
ML



Robin Milner

1987
Haskell



Haskell Committee

2007
C#

Microsoft

2011
C++11

ISO International Organization for Standardization

FP History

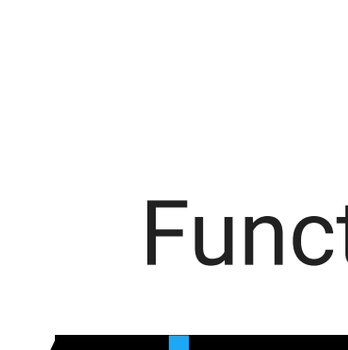
Functional Languages

Mainstream

1936
λ calculus

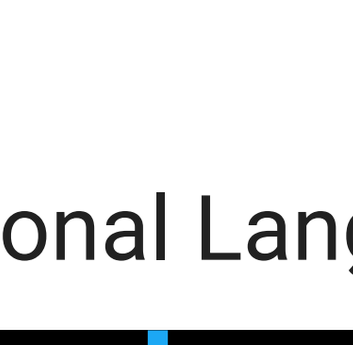
Alonzo Church

1958
Lisp



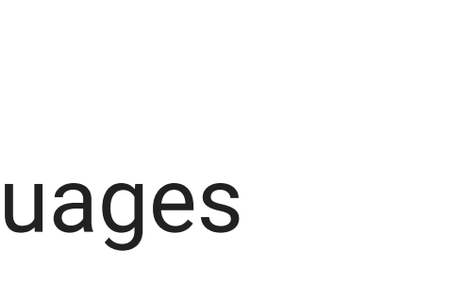
John McCarthy

1973
ML



Robin Milner

1987
Haskell



Haskell Committee

2007
C#

Microsoft

2011
C++11

ISO International Organization for Standardization

2014
Java 8

ORACLE

Swift

Haskell's Unofficial Motto



Simon Peyton Jones
Microsoft Research

Haskell's Unofficial Motto



Simon Peyton Jones
Microsoft Research

Haskell's Unofficial Motto



Simon Peyton Jones
Microsoft Research

Haskell's Unofficial Motto



Avoid Success At All Costs

Pure

Simon Peyton Jones
Microsoft Research

Haskell's Unofficial Motto



Avoid Success At All Costs

Pure | Principled

Simon Peyton Jones
Microsoft Research

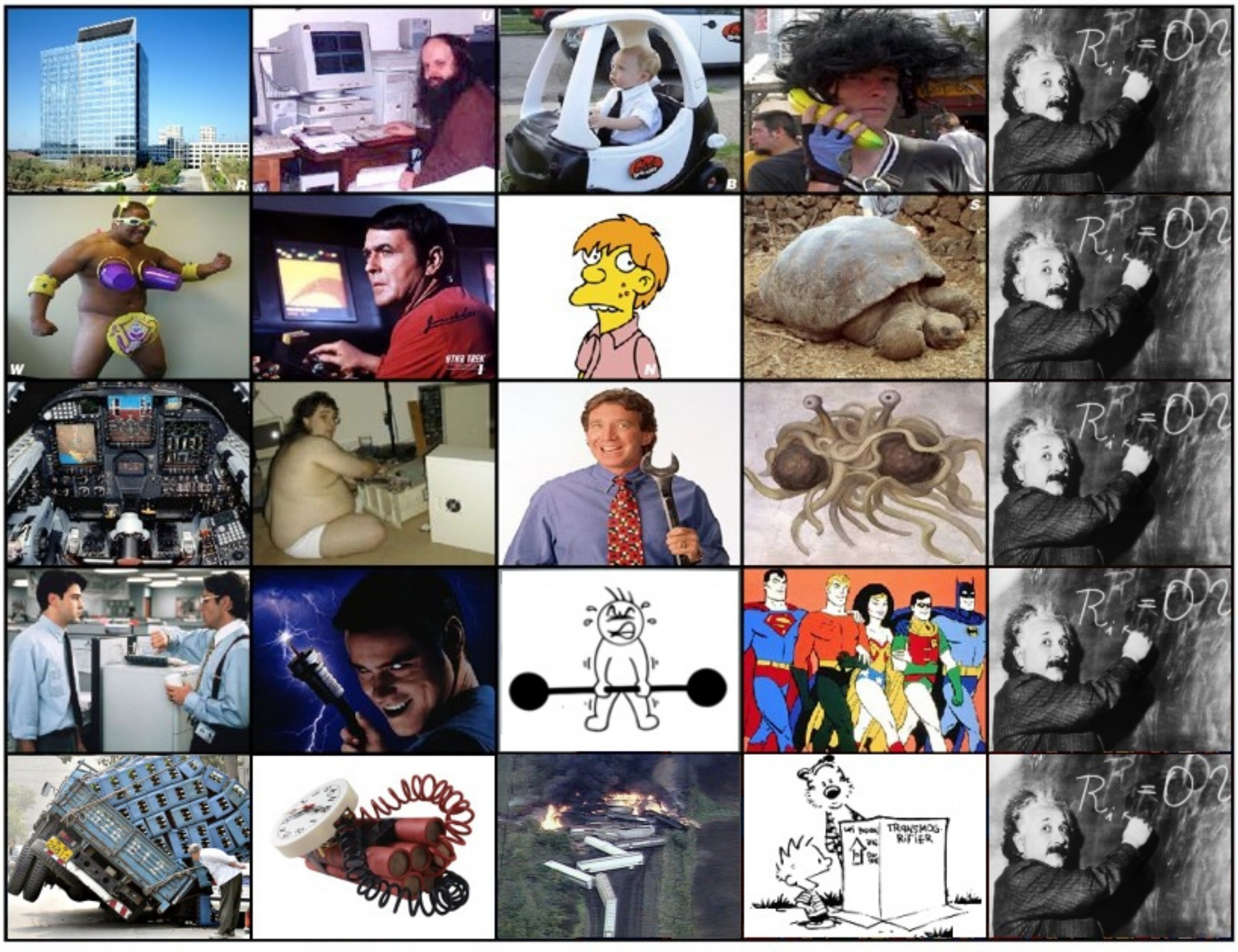
Haskell's Unofficial Motto



Avoid Success At All Costs

Pure | Principled | Nimble

Simon Peyton Jones
Microsoft Research


Java
C
PHP
Ruby
Haskell
as seen by...
Java fans
C fans
PHP fans
Ruby fans
Haskell fans

Haskell in Industry

Finance







Deutsche Bank



Telecom





at&t



Alcatel·Lucent

Many Others








CIJFERS & LETTERS

The Countdown Problem

The Problem

Given

source numbers `[1,3,7,10,25,50]`

target number `765`

The Problem

Given

source numbers `[1,3,7,10,25,50]`

target number `765`

Wanted

expression `(1+50)*(25-10)`

The Problem

Given

source numbers `[1,3,7,10,25,50]`

target number `765`

Wanted

=

expression `(1+50)*(25-10)`

User-Defined Datatype

```
data Op = Add | Sub | Mul | Div
```

User-Defined Datatype

```
data Op = Add | Sub | Mul | Div
```

“enum”-type with 4 alternatives

User-Defined Datatype

```
data Op = Add | Sub | Mul | Div
```

"enum"-type with 4 alternatives

values

User-Defined Datatype

type name

```
data Op = Add | Sub | Mul | Div
```

values

"enum"-type with 4 alternatives

User-Defined Function

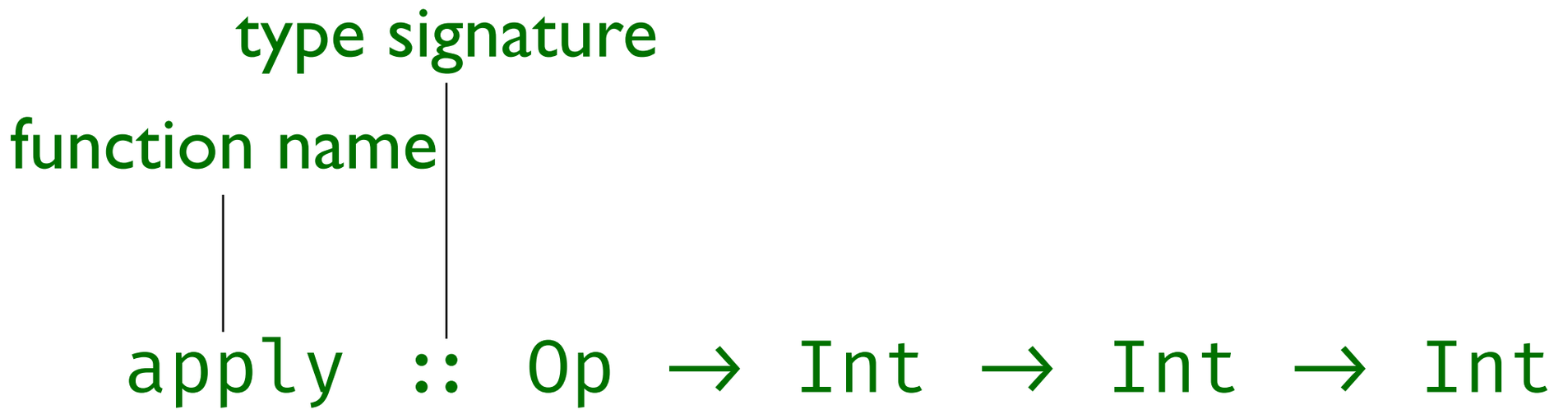
```
apply :: Op → Int → Int → Int
```

User-Defined Function

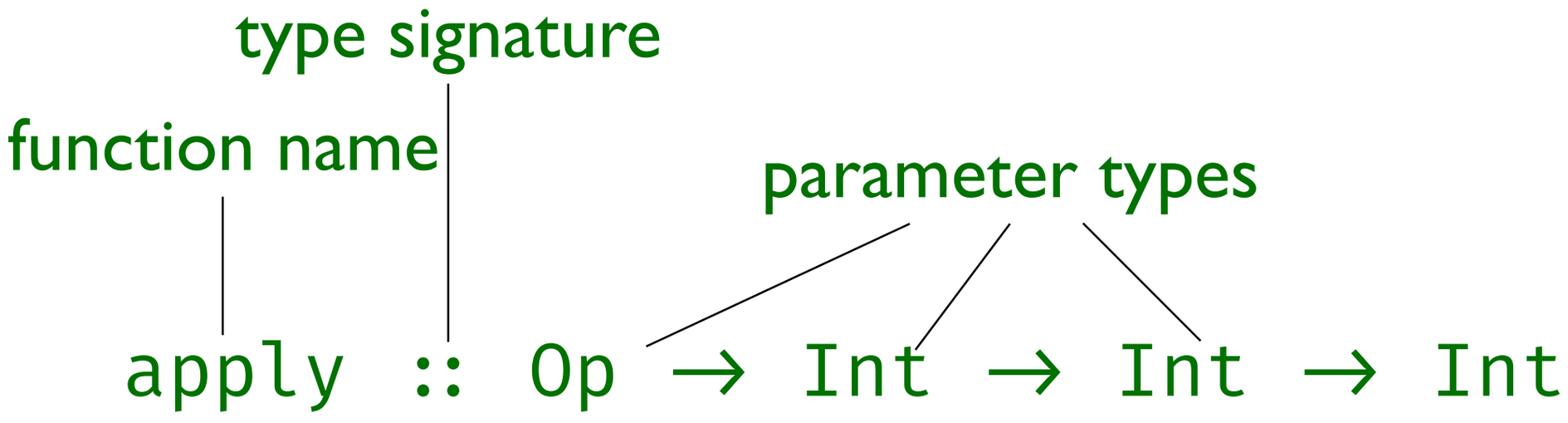
type signature

```
apply :: Op → Int → Int → Int
```

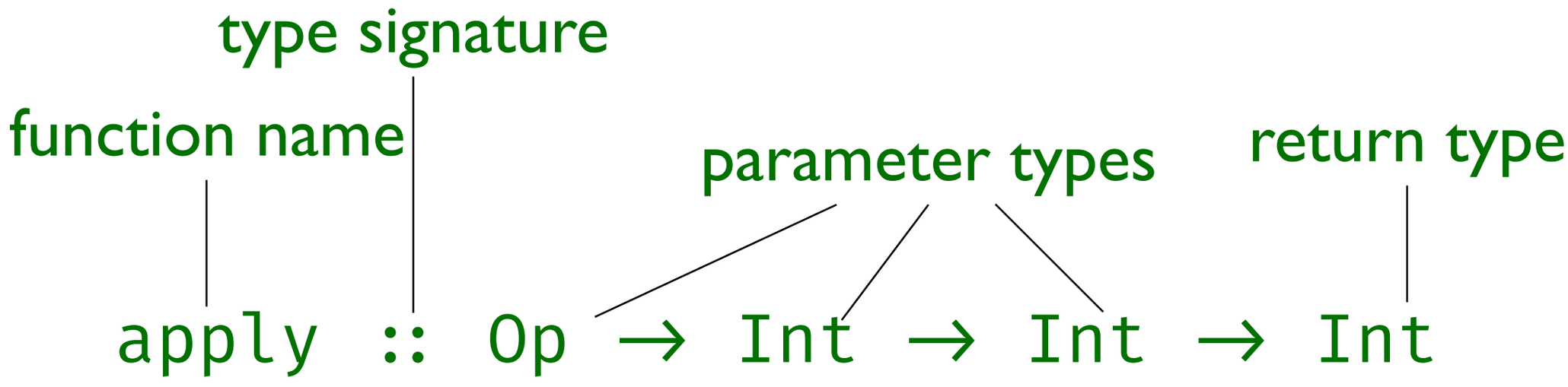
User-Defined Function



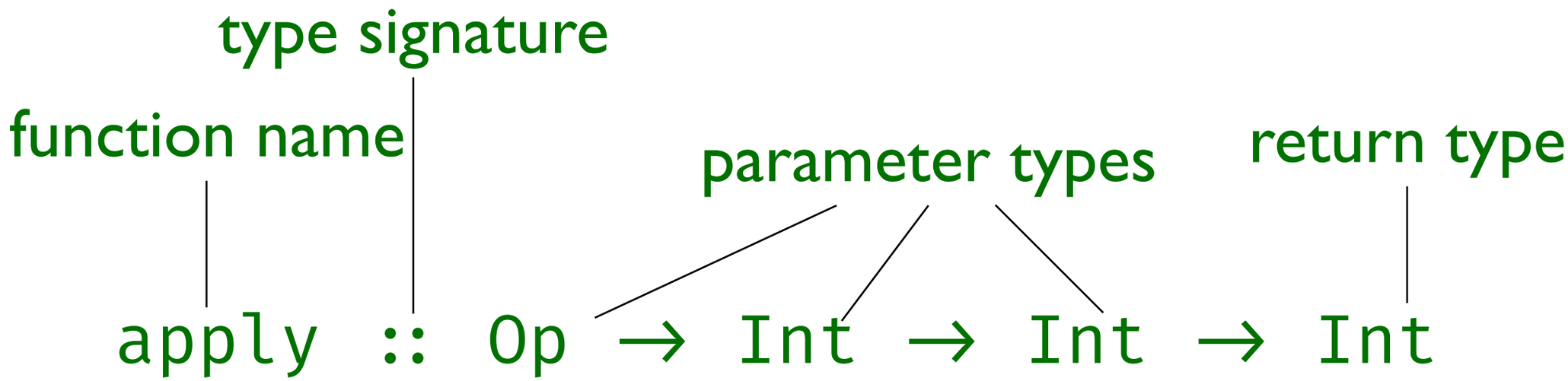
User-Defined Function



User-Defined Function

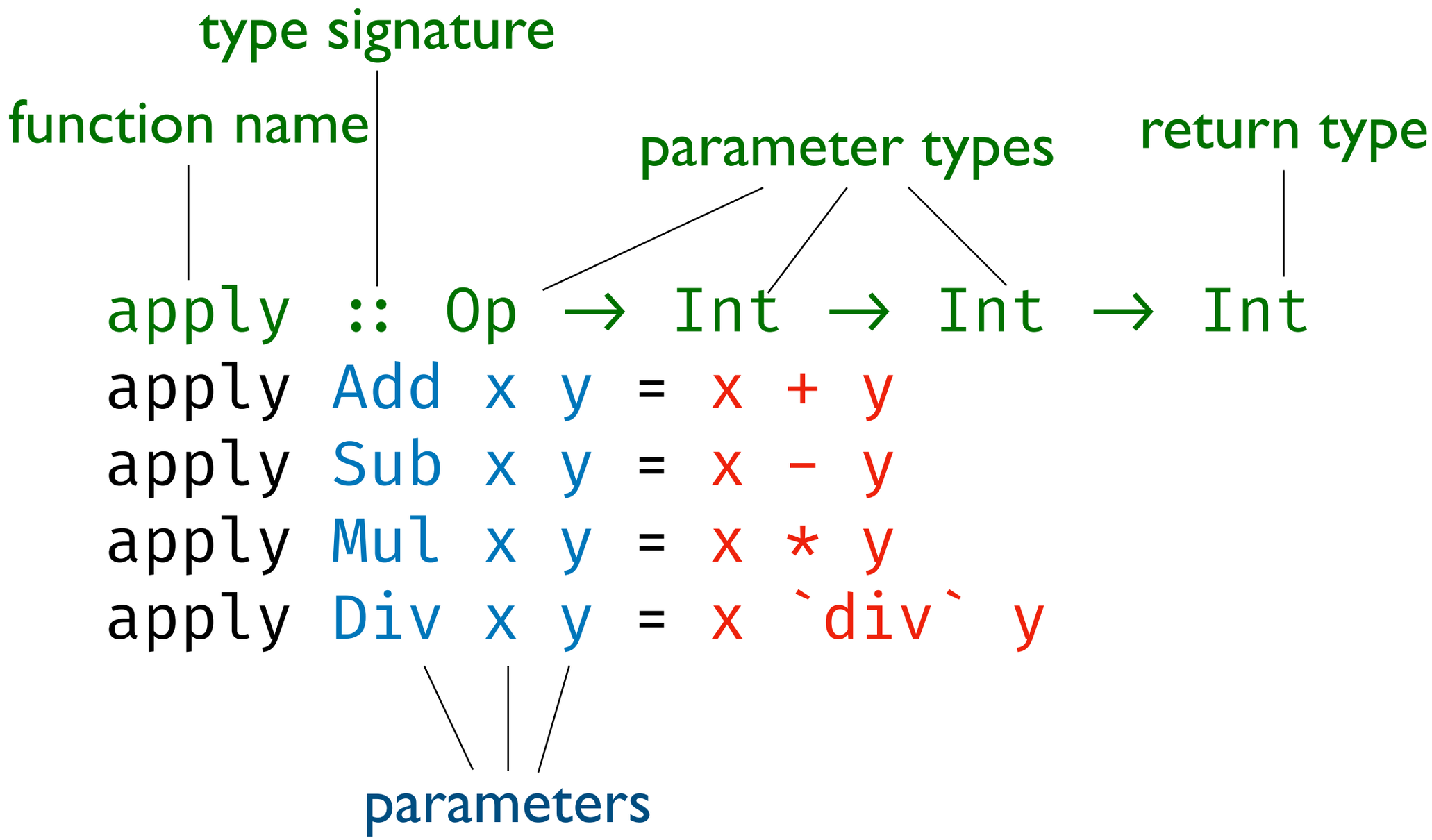


User-Defined Function

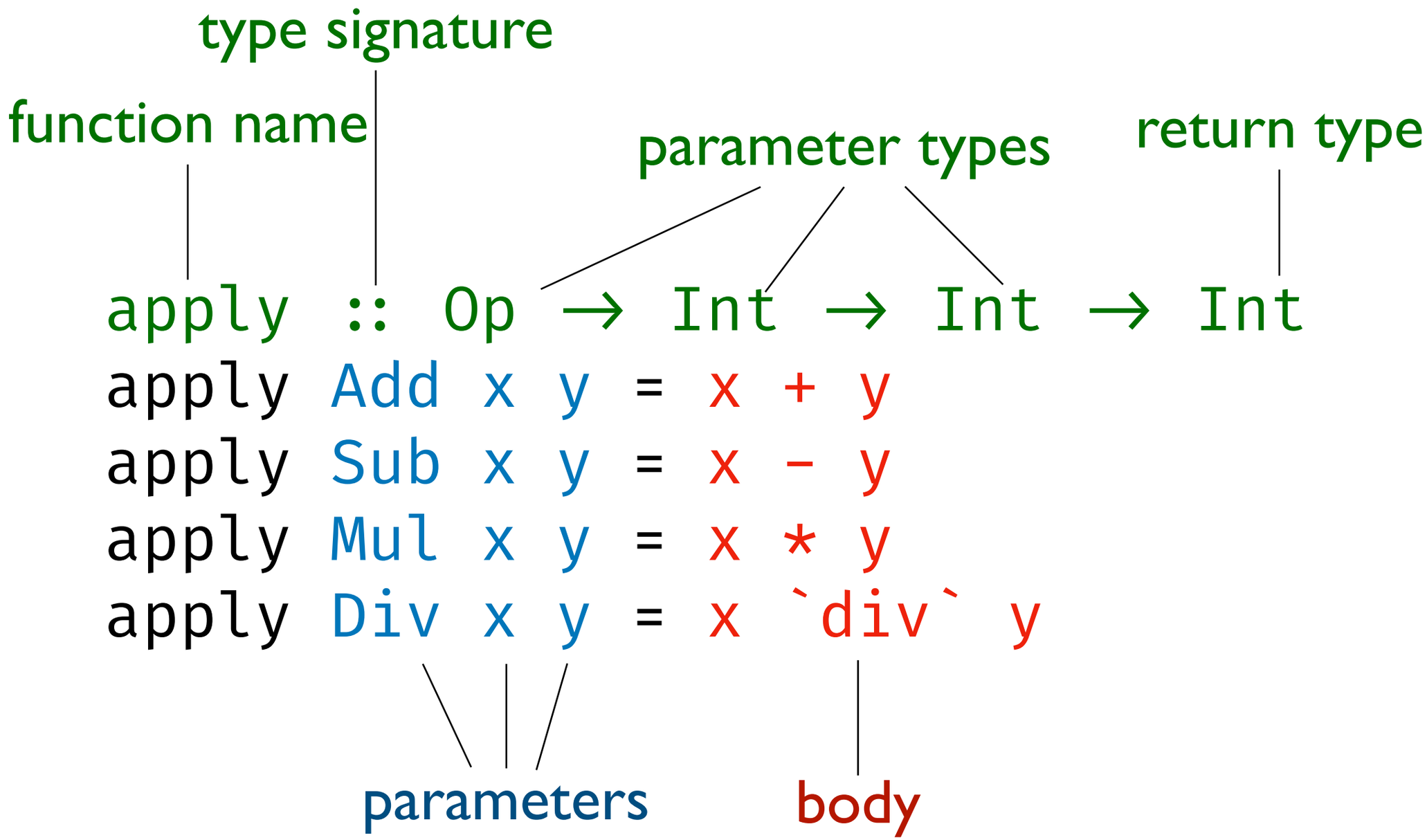



```
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

User-Defined Function



User-Defined Function



User-Defined Function

```
apply :: Op → Int → Int → Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

case analysis
by
pattern matching

GHCi REPL

```
> apply Add 1 2
```

GHCi REPL

```
> apply Add 1 2
3
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
> apply Sub 4 1
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
> apply Sub 4 1
3
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
> apply Sub 4 1
3
> apply Div 3 0
```

GHCi REPL

```
> apply Add 1 2
3
> apply Mul 3 2
6
> apply Sub 4 1
3
> apply Div 3 0
*** Exception: divide by zero
>_
```

Another Function

```
valid :: Op → Int → Int → Bool
valid Add x y = True
valid Sub x y = x > y
valid Mul x y = True
valid Div x y = x `mod` y == 0
```

Recursive Datatype

```
data Expr = Val Int | App Op Expr Expr
```

Recursive Datatype

type name

```
data Expr = Val Int | App Op Expr Expr
```

Recursive Datatype

type name

```
data Expr = Val Int | App Op Expr Expr
```

data constructors

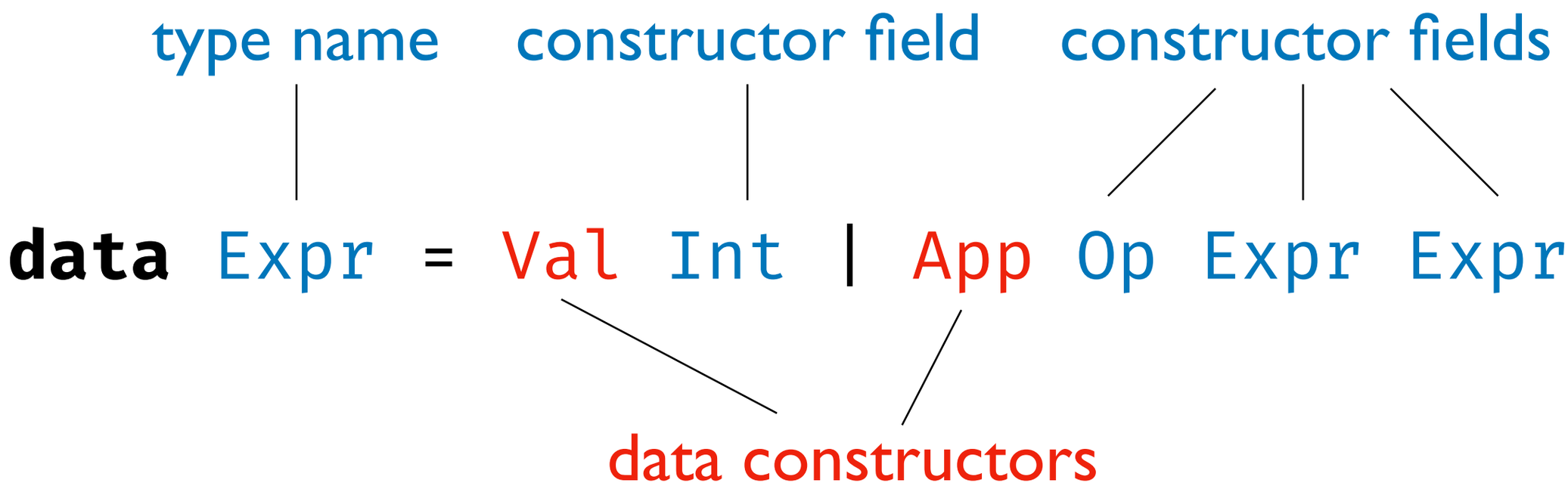
Recursive Datatype

type name constructor field

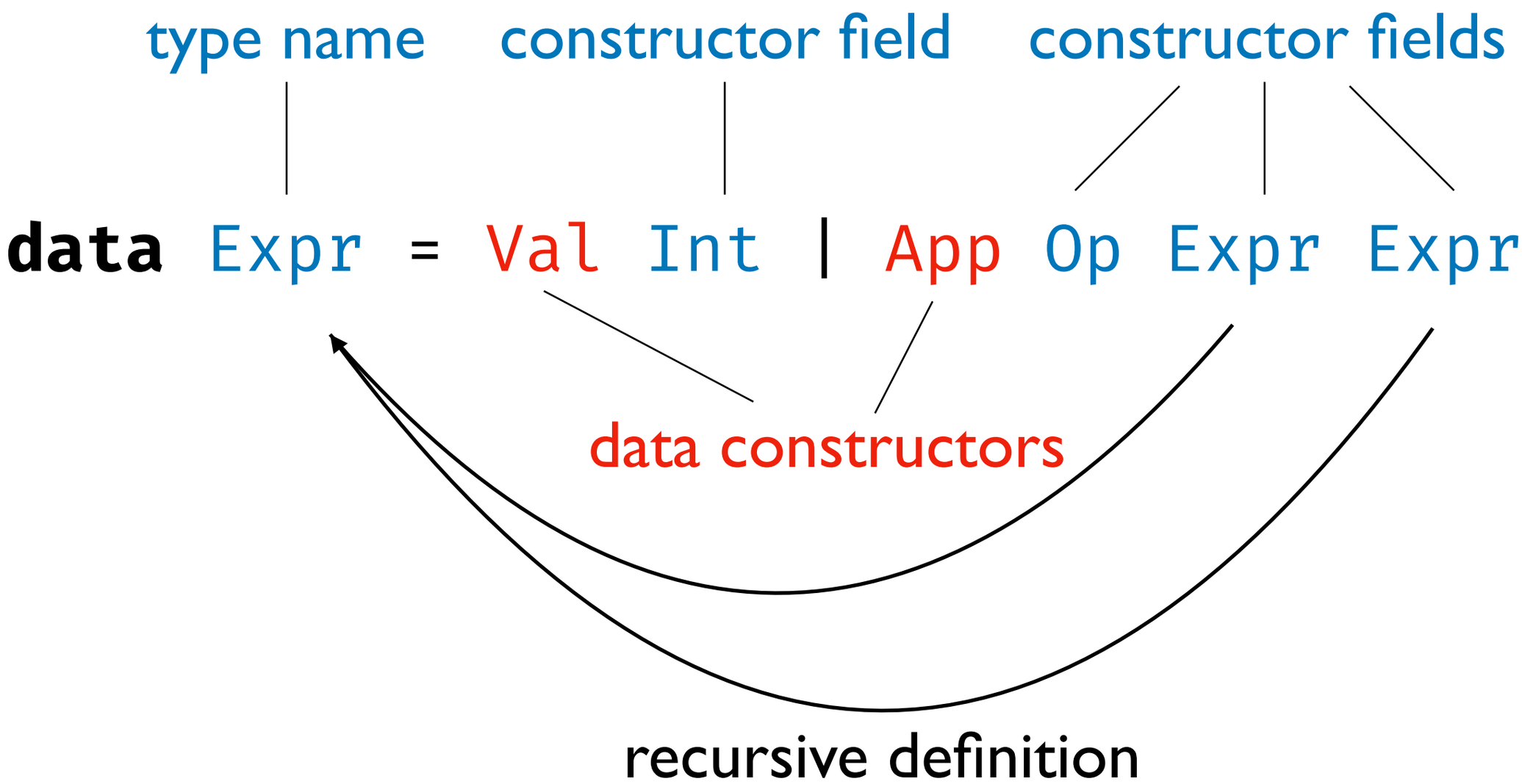
```
data Expr = Val Int | App Op Expr Expr
```

data constructors

Recursive Datatype

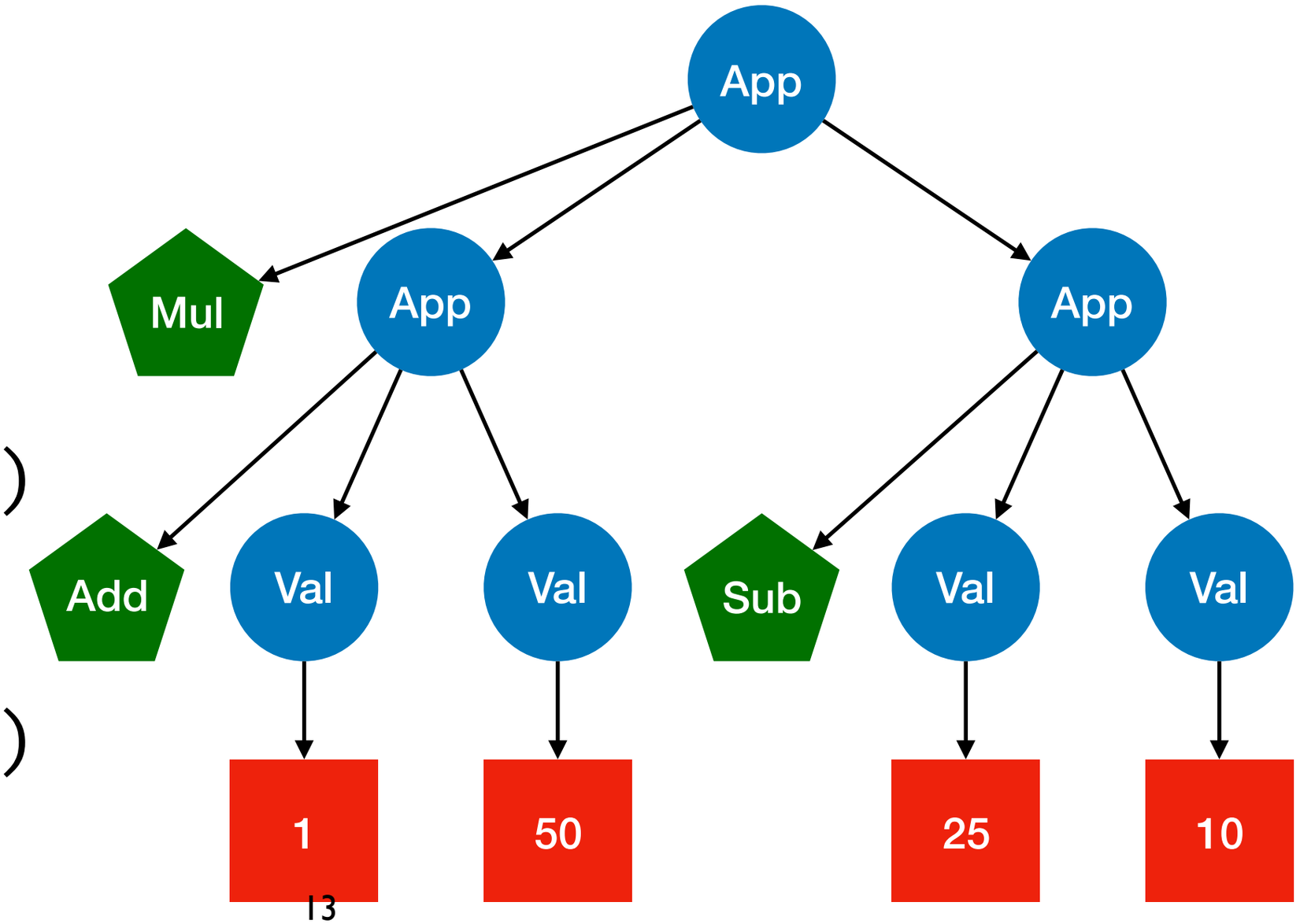


Recursive Datatype



Recursive Datatype

```
example :: Expr
example =
  App Mul
    (App Add
        (Val 1)
        (Val 50))
    (App Sub
        (Val 25)
        (Val 10))
```



Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

```
values :: Expr → [Int]
```

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

```
values :: Expr → [Int]
```

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

```
values :: Expr → [Int]
values (Val n)     = [n]
values (App _ l r) = values l ++ values r
```

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

singleton list

```
values :: Expr → [Int]
values (Val n)      = [n]
values (App _ l r) = values l ++ values r
```

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

```
values :: Expr → [Int]
values (Val n)      = [n]
values (App _ l r) = values l ++ values r
```

singleton list

list concatenation

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

singleton list

list concatenation

```
values :: Expr → [Int]
values (Val n)       = [n]
values (App _ l r) = values l ++ values r
```

recursive definition

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

list of Int

singleton list

list concatenation

```
values :: Expr → [Int]
values (Val n)       = [n]
values (App _ l r) = values l ++ values r
```

recursive definition

function follows recursion of datatype

Recursive Fuction

```
data Expr = Val Int | App Op Expr Expr
```

```
values :: Expr → [Int]
values (Val n)       = [n]
values (App _ l r) = values l ++ values r
```

function follows recursion of datatype

```
> values example
[1,50,25,10]
```

List Comprehension

```
eval :: Expr → [Int]
eval (Val n)      =
  [n | n > 0]

eval (App o l r) =
  [apply o x y
  | x ← eval l
  , y ← eval r
  , valid o x y
  ]
```

List Comprehension

```
eval :: Expr → [Int]
eval (Val n)      =
  [n | n > 0]
eval (App o l r) =
  [apply o x y
  | x ← eval l
  , y ← eval r
  , valid o x y
  ]
```

generators

List Comprehension

```
eval :: Expr → [Int]
eval (Val n)       =
  [n | n > 0]
eval (App o l r) =
  [apply o x y
  | x ← eval l
  , y ← eval r
  , valid o x y
  ]
```

guards

generators

List Comprehension

```
eval :: Expr → [Int]
eval (Val n)       =
  [n | n > 0]
eval (App o l r) =
  [apply o x y
  | x ← eval l
  , y ← eval r
  , valid o x y
  ]
```

guards

generators

```
> eval example
[765]
```

Parametric Polymorphism

```
subbags :: [a] → [[a]]
subbags xs = [zs | ys ← subs xs
                 , zs ← perms ys ]
```

Parametric Polymorphism

type variable

```
subbags :: [a] → [[a]]
subbags xs = [zs | ys ← subs xs
                 , zs ← perms ys ]
```

Parametric Polymorphism

aka
Generics

type variable

```
subbags :: [a] → [[a]]
subbags xs = [zs | ys ← subs xs
                 , zs ← perms ys ]
```

Parametric Polymorphism

aka
Generics

type variable

```
subbags :: [a] → [[a]]
subbags xs = [zs | ys ← subs xs
                 , zs ← perms ys ]
```

```
> subbags [1,2]
[[],[1],[2],[1,2],[2,1]]
> subbags [True,False]
[[],[True],[False],[True,False],[False,True]]
```

Check Solution

```
solution :: Expr → [Int] → Int → Bool
solution e ns n =
      elem (values e) (subbags ns)
   && eval e == [n]
```

Check Solution

```
solution :: Expr → [Int] → Int → Bool
solution e ns n =
      elem (values e) (subbags ns)
   && eval e == [n]
```

```
> solution example [1,3,7,10,25,50] 765
True
```

Generate Solution

```
> split [1,2,3]
[([],[1,2,3]),([1],[2,3]),([1,2],[3])
,([1,2,3],[])]
```

Generate Solution

```
> split [1,2,3]
[([],[1,2,3]),([1],[2,3]),([1,2],[3])
,([1,2,3],[])]
```

```
split :: [a] → [([a],[a])]
```

Generate Solution

```
> split [1,2,3]
[([],[1,2,3]),([1],[2,3]),([1,2],[3])
,([1,2,3],[])]
```

```
split :: [a] → [([a],[a])]
split []     = [([],[])]
```

Generate Solution

```
> split [1,2,3]
[([],[1,2,3]),([1],[2,3]),([1,2],[3])
,([1,2,3],[])]
```

```
split :: [a] → [([a],[a])]
split []     = [([],[])]
split (x:xs) =
  ([],x:xs) : [(x:ls,rs)
              |(ls,rs)←split xs]
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

```
ne :: ([a],[a]) → Bool
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

```
ne :: ([a],[a]) → Bool
ne (xs,ys) = not (null xs || null ys)
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

```
ne :: ([a],[a]) → Bool
ne (xs,ys) = not (null xs || null ys)

nesplit :: [a] → [([a],[a])]
```

Generate Solution

```
> nesplit [1,2,3]
[([1],[2,3]),([1,2],[3])]
```

```
ne :: ([a],[a]) → Bool
ne (xs,ys) = not (null xs || null ys)

nesplit :: [a] → [([a],[a])]
nesplit l = [t | t ← split l, ne t]
```

Generate Solution

```
combine :: Expr → Expr → [Expr]
combine l r = [App o l r | o ← ops]

ops :: [Op]
ops = [Add,Sub,Mul,Div]
```

Generate Solution

```
exprs :: [Int] → [Expr]
exprs []  = []
exprs [n] = [Val n]
exprs ns  = [e | (ls,rs) ← nesplit ns
               , l ← exprs ls
               , r ← exprs rs
               , e ← combine l r
               ]
```

Solutions

```
solutions :: [Int] → Int → [Expr]
solutions ns n
  = [e | ns' ← subbags ns
       , e   ← exprs ns'
       , eval e == [n]
       ]
```

Functional Pearl: The countdown problem.
Graham Hutton. Journal of Functional Programming.
2002.

Solutions

```
solutions :: [Int] → Int → [Expr]
solutions ns n
  = [e | ns' ← subbags ns
       , e   ← exprs ns'
       , eval e == [n]
       ]
```

naive
generate & test
strategy

Functional Pearl: The countdown problem.
Graham Hutton. Journal of Functional Programming.
2002.

... lo esso: digamos el meior.
... q̃ caye en precio: del sc̃o confessor.
... se aduzir: este ciego lazrado.
... casa del monge: de suso ementado.
... creye bien afirmes: estaua fyuzado.
... desta coyta: por elli terminado.
... fue ala puerta: de sanct sabastian.
... quiso el mesquino: pedir uino ni pan.
... dicie ay padre: por señor san ...
... prenda cordoio: de este mi affan.
... alla do yaces yo a ti bin buscar.
... tu ... amu alla tornar
... yo non podria: partirme de ...
... tu me mandes: seer otorn...
... delos lazrados: deña ...
... sobre mi tu mano: signa me del polgar.
... q̃ yo pudiesse: la tu mano besar.
... esta coyta: cuydaria sanar
... padre beneyto: bien entro do estaua.
... los apellidos: q̃ este ciego daua.
... e pregunto le: q̃l cosa demandaua.
... el q̃ hurne: ca el non cobdiciaua.
... sc̃o ...

Echol con el ysopo: del agua salada.
Consigno li los oios con la cruz consagrada.
La dolor e la coyta: fue luego amansada.
La lumine q̃ perdiera: fue toda recobrada.
Entenderlo pudiestes: amigos e señores.
E auie muchos males: de diuersos colores.
Vnos de ceguedad: al de graues dolores.
Mas de todo bien sano: rendie a dios lodores.
Oreo el padre sc̃o: amigo ue tu uia.
Gradecelo a dios: q̃ uas com meioria.
C... q̃ no ... e no fagas follia.
... tu ... faces recadia.
... los ... q̃ dest padre sabemos.
... os unos o oymos: los otros q̃ leemos.
En dubda nos paramos: en q̃l enpeçaremos.
Mas aq̃l parte q̃ sea: adelhar aruremos.
Desta sazon los otros: q̃ero los fer esquiuos.
Dezir uno e miembre uos: mientre fueredes uiuos.
Como gano la gra: q̃ saca los catiuos.
Por ond de luengas tierras: le enbian bodigos.
Era en essi tiempo: los moros muy uezinos.

el pmer ro cati uo q̃ saco en su uida.

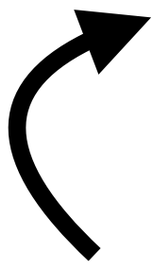
DSLs

DSL

=

Domain Specific Language

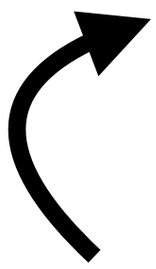
Conventional Approach



1. solve a problem with a general-purpose language
2. $

DSL Approach

1. develop a language



2. solve problems with that language

3. $$$

Develop a Language

★ *language of discourse*

★ *declarative*
(not operational/imperative)

★ domain-specific: facilitate expressing relevant concepts

★ compositional / functional / bottom-up

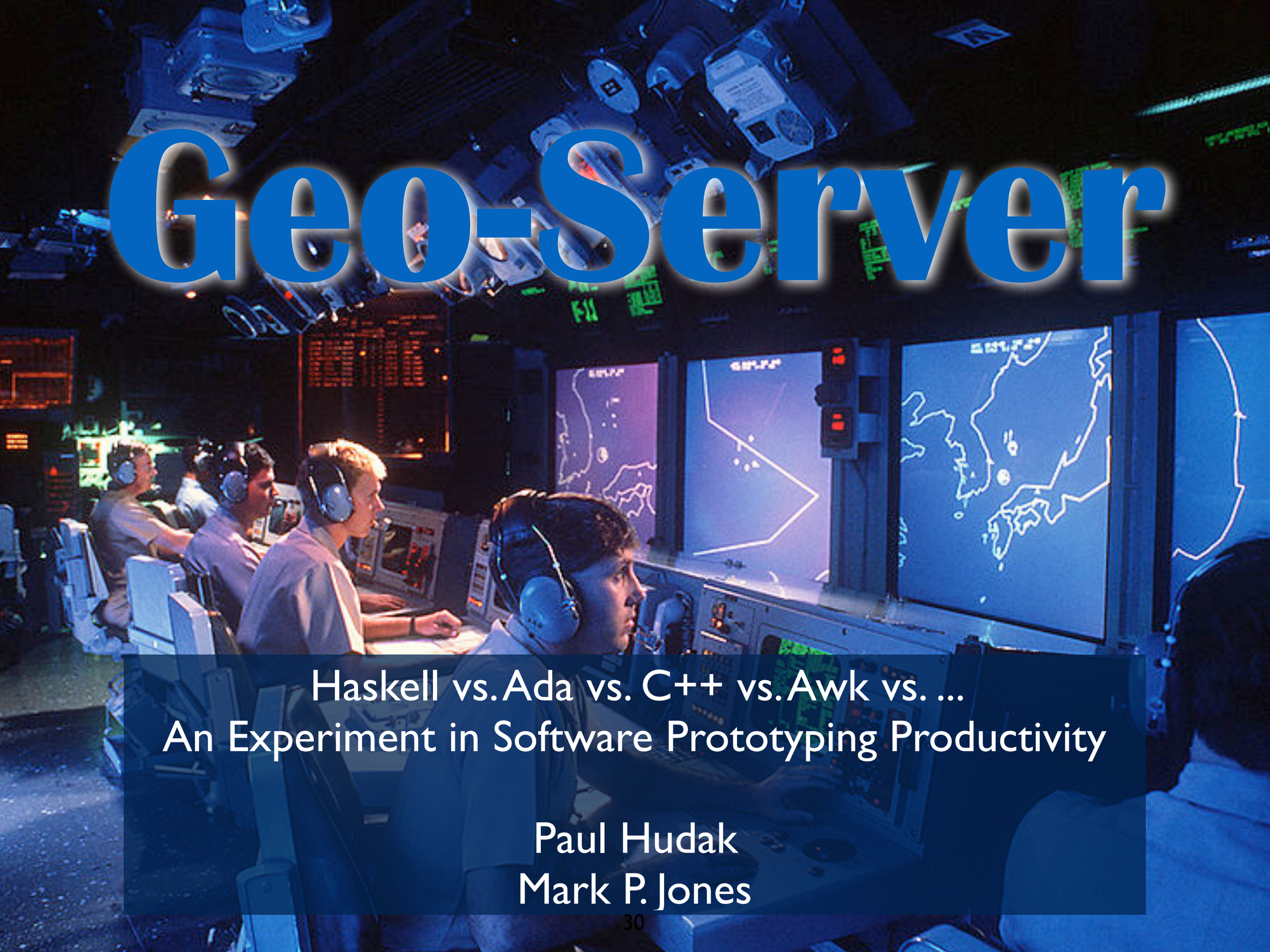
Develop a Language

- ★ executable software artefact
  - ✦ refinable, but actual part of the final solution
- ★ multiple meanings/modes of execution
  - ✦ tests, simulations, platforms, parallelism
  - ✦ customisation

Types of DSLs

- Stand-alone DSL
  HTML, Verilog, SQL, YACC, GraphViz, ...
- Embedded DSL (EDSL)
  embedded in a host language like
  Haskell as library
  (but also in Scala, Groovy, …)

Geo-Server



Haskell vs. Ada vs. C++ vs. Awk vs. ...
An Experiment in Software Prototyping Productivity

Paul Hudak
Mark P. Jones

The Setup

- US Navy Experiment
- Study suitability of languages for rapid prototyping
- Languages: Haskell, Ada, Ada9X, C++, Awk, Rapide, Griffin, Proteus, Relational Lisp
- one expert programmer, small project



Problem

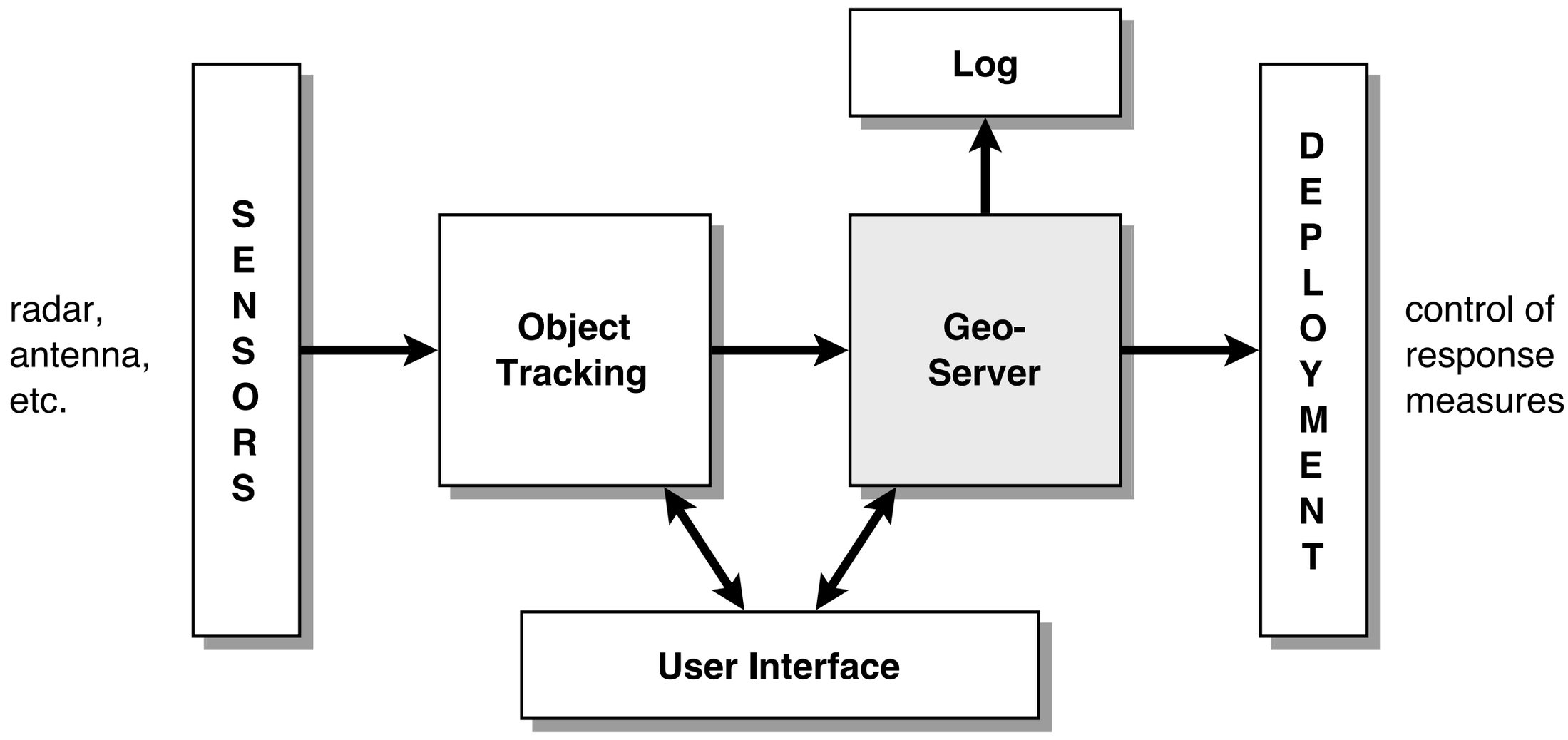


Figure 1: Simplified Aegis Weapons Systems Diagram

Geo-Server Input

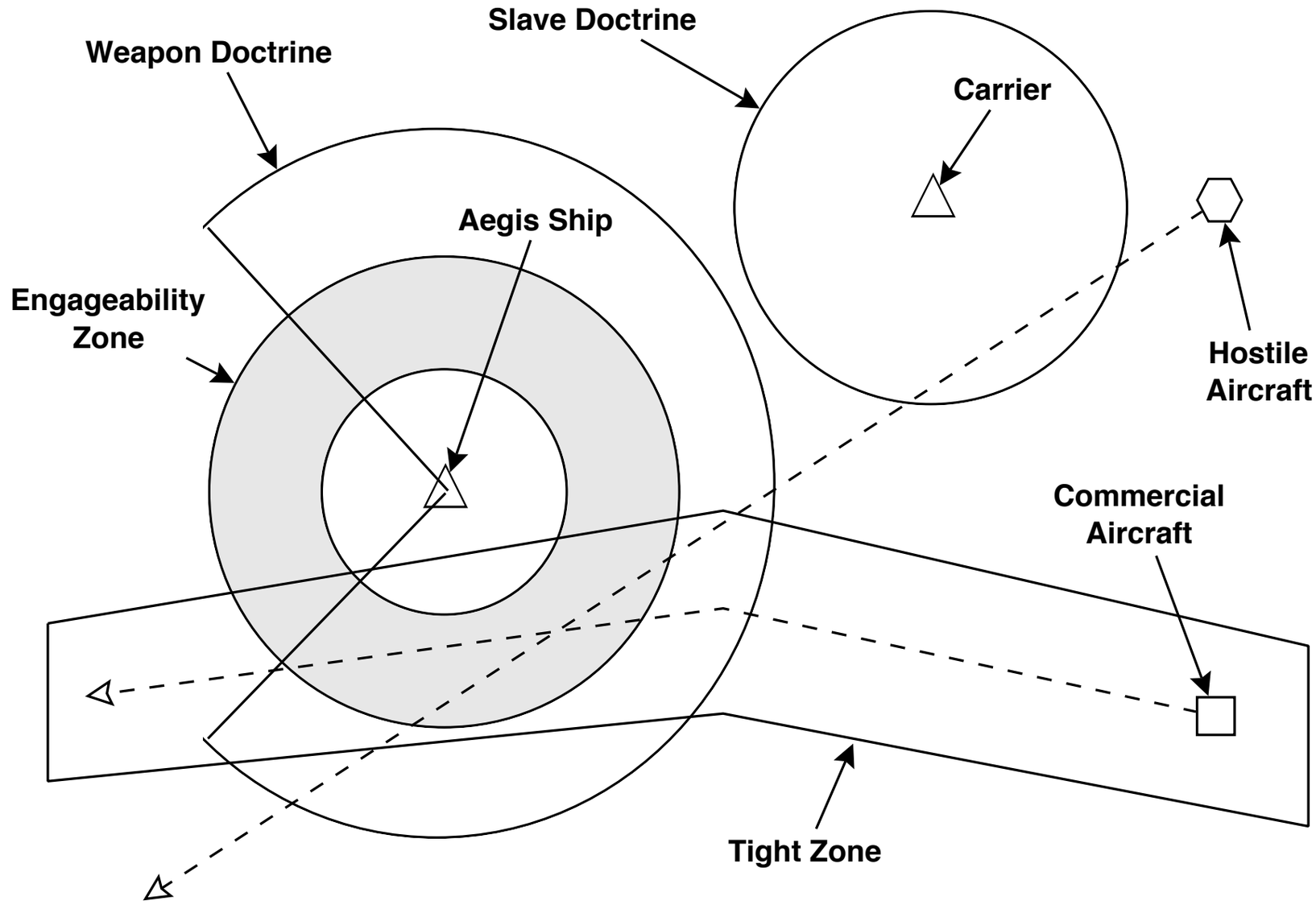


Figure 2: Geo-Server Input Data

Geo-Server Output

```
Time 0.0:
commercial aircraft: (38.0,25.0)
 -- In tight zone
hostile craft: (258.0,183.0)

Time 20.0:
commercial aircraft: (58.0,30.0)
 -- In tight zone
hostile craft: (239.0,164.0)

Time 40.0:
commercial aircraft: (100.0,43.0)
 -- In engageability zone
 -- In tight zone
hostile craft: (210.0,136.0)
 -- In carrier slave doctrine
```

Haskell Solution

```
type Region

inRegion :: Point  → Region → Bool
circle   :: Radius → Region
outside  :: Region → Region
(/\)     :: Region → Region → Region
```

Haskell Solution

```
type Region

inRegion :: Point  → Region → Bool
circle   :: Radius → Region
outside  :: Region → Region
(/\)     :: Region → Region → Region
```

```
annulus :: Radius → Radius → Region
annulus r1 r2 = outside (circle r1) /\
                circle r2
```

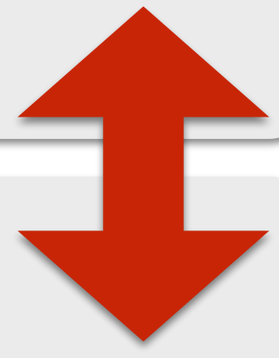
Possible Implementations

Shallow embedding

★ implement combinators as functions

★ semantics: `inRegion`

★ no "interpretative overhead"



Deep embedding

★ implement combinators as a datatype

★ multiple ways to handle regions:
`inRegion, print, simplify,...`

Shallow Embedding

```
type Region = Point → Bool

p `inRegion` r  = r p

circle d  = \p → distance (0,0) p ⩽ d

outside r = \p → not (r p)

r1 /\ r2  = \p → r1 p && r2 p
```

Deep Embedding

```
type Region = R

data R = Circle Radius
       | Outside R
       | Intersect R R
```

Deep Embedding

```
type Region = R

data R = Circle Radius
       | Outside R
       |Intersect R R
```

```
circle  = Circle
outside = Outside
(/\)    = Intersect
```

Deep Embedding

```haskell
type Region = R

data R = Circle Radius
       | Outside R
       |Intersect R R
```

```haskell
circle  = Circle
outside = Outside
(/\)    = Intersect
```

```haskell
p `inRegion` (Circle d)
  = distance (0,0) p ⩽ d
p `inRegion` (Outside r)
  = not (p `inRegion` r)
p `inRegion` (Intersect r1 r2)
  = (p `inRegion` r1) && (p `inRegion` r2)
```

Smart Constructors

```
type Region = R

data R = Circle Radius
       | Outside R
       | Intersect R R
```

Smart Constructors

```haskell
type Region = R

data R = Circle Radius
       | Outside R
       |Intersect R R
```

```haskell
circle  = opt . Circle
outside = opt . Outside
(/\)    = …
```

Smart Constructors

```
type Region = R

data R = Circle Radius
       | Outside R
       |Intersect R R
```

```
circle  = opt . Circle
outside = opt . Outside
(/\)    = …
```

```
opt :: R → R
opt (Intersect (Circle d1) (Circle d2))
  = Circle (max d1 d2)
 …
opt r = r
```

Study Results

Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

Financial Contracts

Financial Contracts

Composing contracts: an adventure in financial engineering

Simon Peyton Jones, Jean-Marc Eber, Julian Seward

Example Contract

The owner of the contract contract has the right to choose on June 30 2000 between:

D_1 Both of:

- D_{11} Receive £100 on 29 Jan 2001.
- D_{12} Pay £105 on 1 Feb 2002.

D_2 An option exercisable on 15 Dec 2000 to choose one of:

- D_{21} Both of:
  - D_{211} Receive £100 on 29 Jan 2001.
  - D_{212} Pay £106 on 1 Feb 2002.
- D_{22} Both of:
  - D_{221} Receive £100 on 29 Jan 2001.
  - D_{222} Pay £112 on 1 Feb 2003.

Problems

- Inaccurate, non-uniform language
- Analysis and manipulation of contracts
  - calculate worth
  - simulate

Simple Contract

```
-- receive £100 on 13/02/2003
c1 :: Contract
c1 = zcb t1 100 GBP

-- zero coupon bond
zcb :: Date → Double → Currency → Contract

mkDate :: String → Date

t1 :: Date
t1 = mkDate “0800GMT 13 Feb 2003”
```

Composing Contracts

```
and :: Contract → Contract → Contract
```

```
c2,c3 :: Contract
c2 = zcb t2 200 GBP
c3 = c1 `and` c2
```

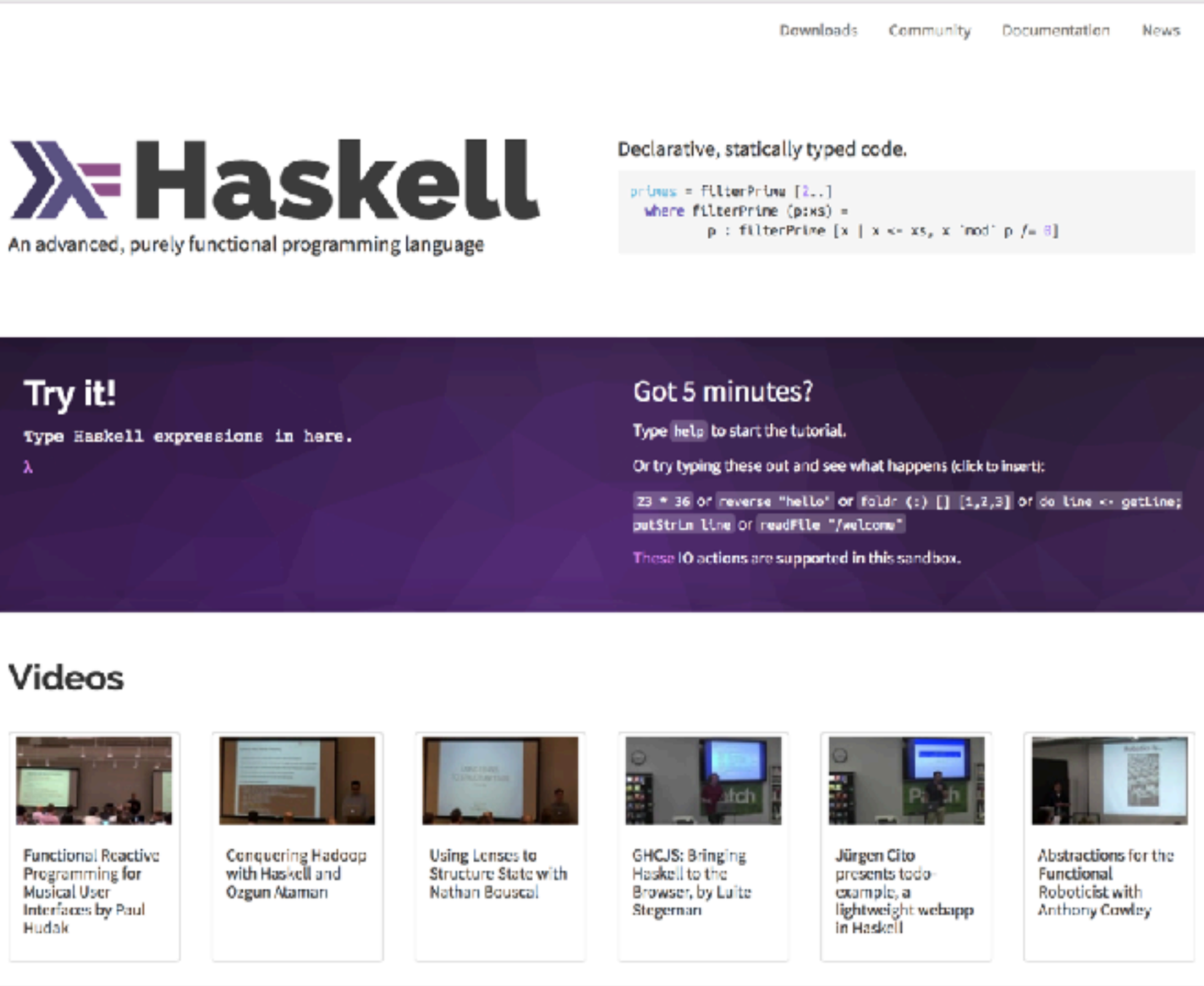
```
give :: Contract → Contract
```

```
andGive :: Contract → Contract → Contract
andGive c d = c `and` give d
```

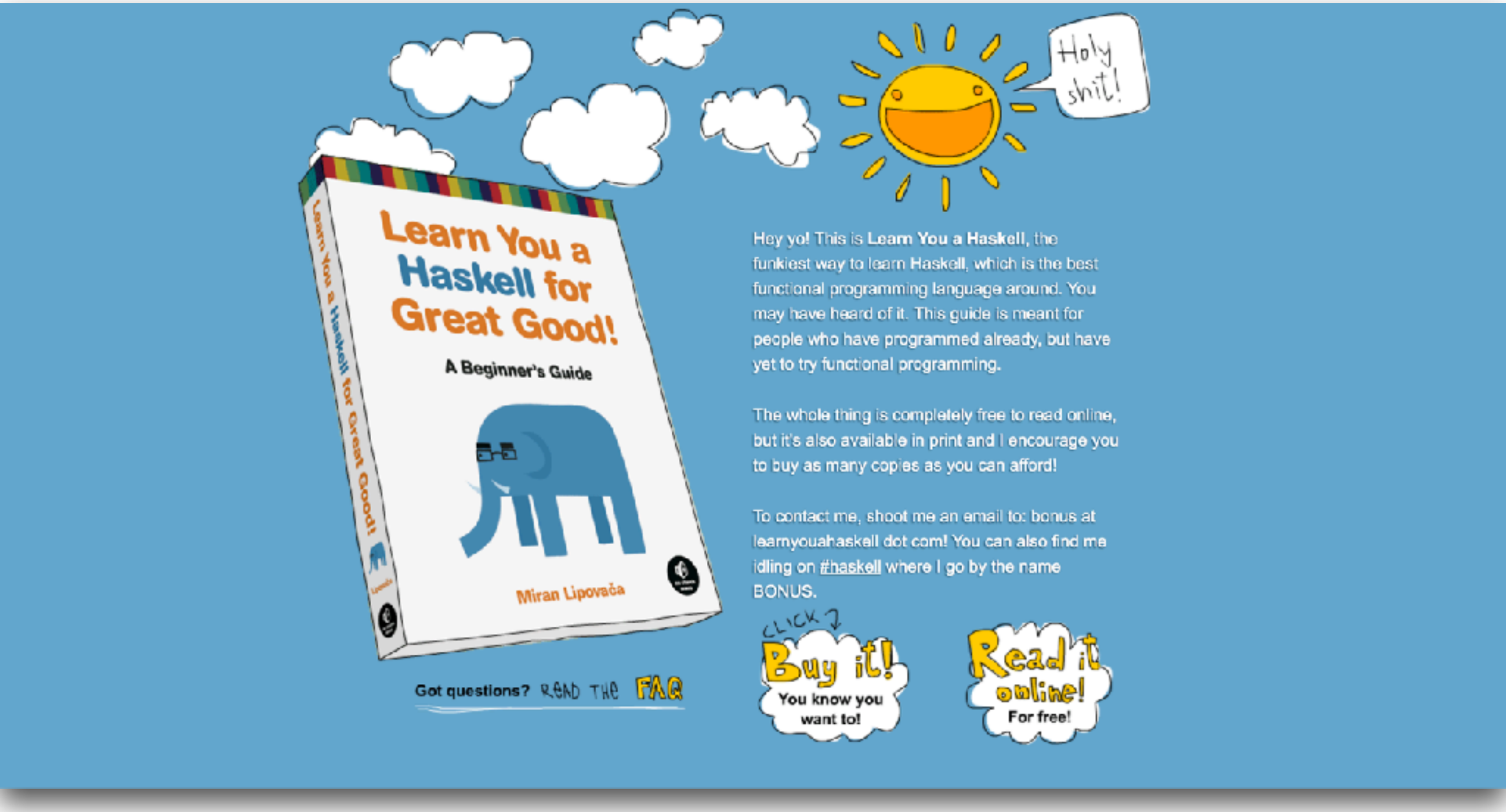
```
c4 = c1 `andGive` c2
```

Resources

haskell.org


Downloads
Community
Documentation
News
Haskell
An advanced, purely functional programming language
Declarative, statically typed code.
primes = filterPrime [2..]
where filterPrime (p:xs) =
p : filterPrime [x | x <- xs, x `mod` p /= 0]
Try it!
Type Haskell expressions in here.
λ
Got 5 minutes?
Type help to start the tutorial.
Or try typing these out and see what happens (click to insert):
23 * 36 or reverse "hello" or foldr (:) [] [1,2,3] or do line <- getLine; putStrLn line or readFile "/welcome"
These IO actions are supported in this sandbox.
Videos
Functional Reactive Programming for Musical User Interfaces by Paul Hudak
Conquering Hadoop with Haskell and Ozgun Ataman
Using Lenses to Structure State with Nathan Bouscal
GHCJS: Bringing Haskell to the Browser, by Luite Stegeman
Jürgen Cito presents todo example, a lightweight webapp in Haskell
Abstractions for the Functional Roboticist with Anthony Cowley

Free Online Book



Free Online Book

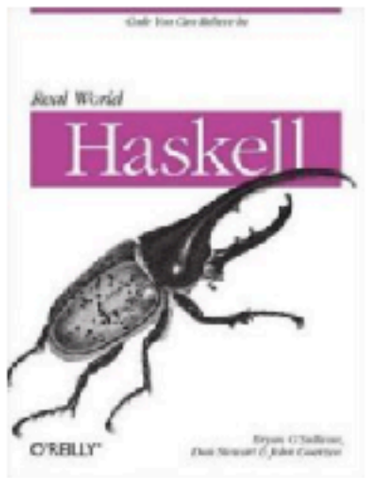
Real World Haskell
by Bryan O'Sullivan, Don Stewart, and John Goerzen

Welcome to Real World Haskell

This is the online home of the book "Real World Haskell". It is published by O'Reilly Media. The first edition was released in November 2008.

We make the content freely available online. If you like it, please buy a copy.

For news updates, please visit our blog.



Buy online

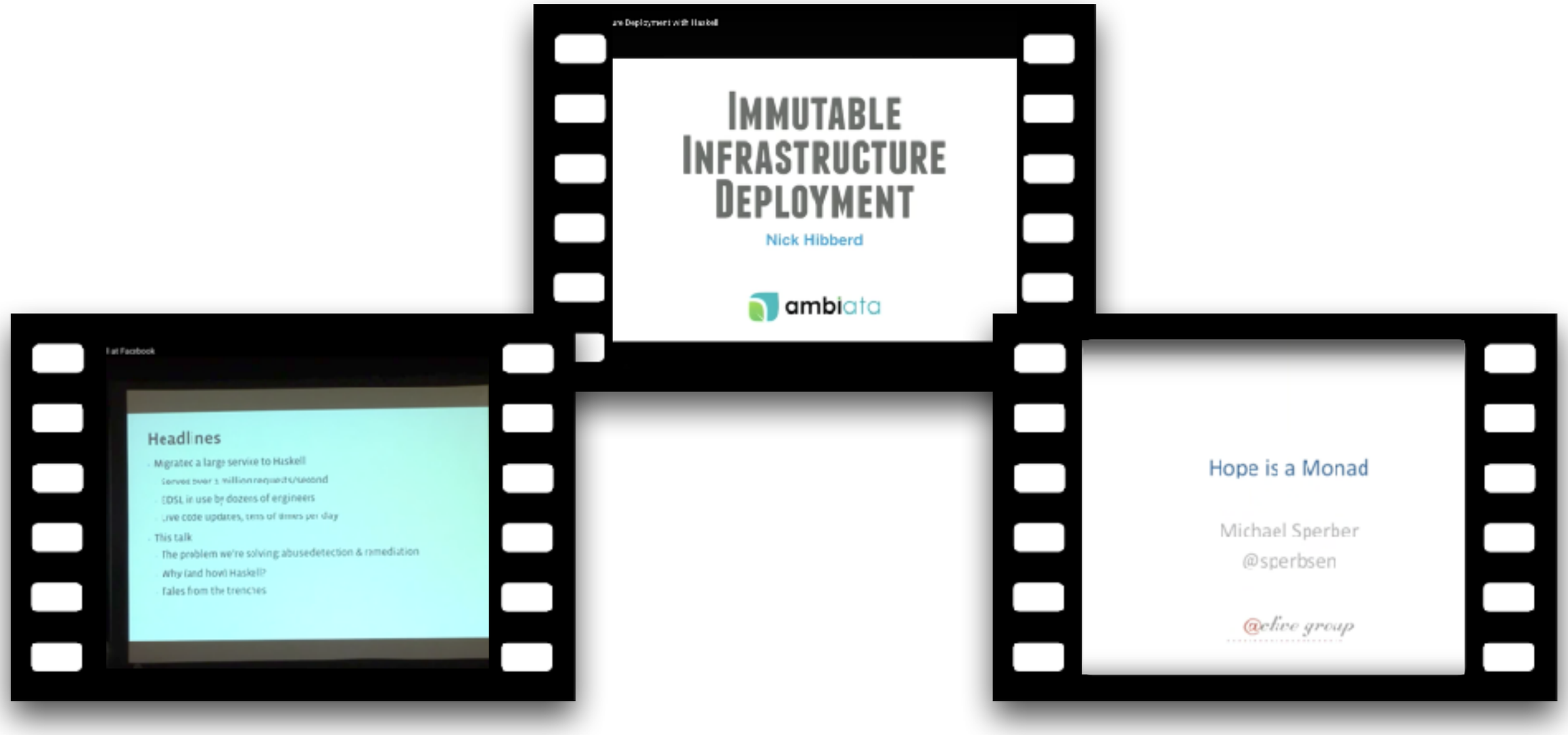
For your convenience, we have included links to the book through sellers in several countries.

- O'Reilly (USA, UK)
- Powell's Books (USA)
- Amazon (USA)
- Amazon (Deutschland)
- Amazon (Canada)
- Amazon (UK)

What's that creature on the cover?

Annual CUFP Workshop

http://cufp.org



Lots of Testimonial Videos

Haskell Support / Consultants







Summary

Haskell

★ Pure Functional Language

★ Avoids Success at All Costs:

- ✦ many industrial users
- ✦ powerful DSLs
- ✦ > 11,600 software packages available on hackage.org

The End