

Taming effects

The next big challenge

Simon Peyton Jones

Microsoft Research

2008

Summary

c.f. static
types 1995-
2005

1. Over the next 10 years, the software battleground will be

the control of **effects**

2. To succeed, we must shift programming perspective

from
Imperative by default
to
Functional by default

Any
effect

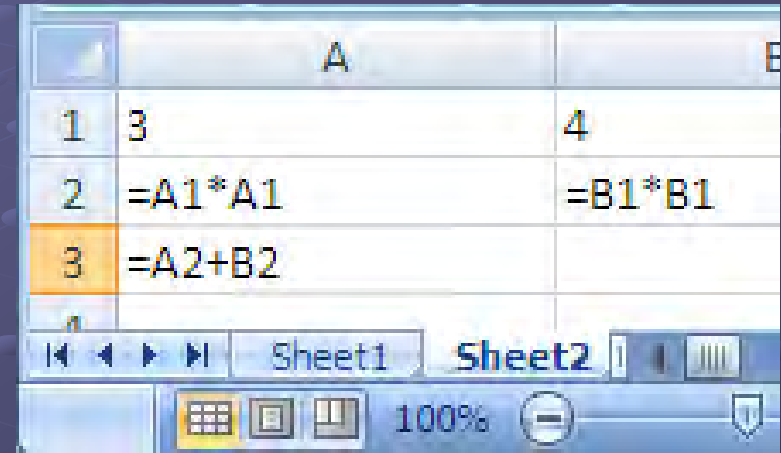
Spectrum

Pure
(no effects)

C, C++, Java, C#, VB

Excel, Haskell

```
X := ln1
X := X*X
X := X + ln2*ln2
```



The screenshot shows an Excel spreadsheet with two columns, A and B, and three rows of data. Row 1 contains the values 3 and 4. Row 2 contains the formulas =A1*A1 and =B1*B1. Row 3 contains the formula =A2+B2. The spreadsheet interface includes a formula bar, sheet tabs for Sheet1 and Sheet2, and a status bar showing 100% zoom.

	A	B
1	3	4
2	=A1*A1	=B1*B1
3	=A2+B2	

Commands, control flow

- Do this, then do that
- “X” is the name of a cell that has different values at different times

Expressions, data flow

- No notion of sequence
- “A2” is the name of a (single) value

Imperative

C, C++, Java, C#, VB



```
X := In1  
X := X * X  
X := X + In2 * In2
```

In1 3

In2 4

X

Commands, control flow

- Do this, then do that
- “X” is the name of a cell that has different values at different times

Imperative

C, C++, Java, C#, VB



```
X := In1  
X := X*X  
X := X + In2*In2
```

In1 3

In2 4

X 3

Commands, control flow

- Do this, then do that
- “X” is the name of a cell that has different values at different times

Imperative

C, C++, Java, C#, VB



```
X := In1  
X := X*X  
X := X + In2*In2
```

In1 3

In2 4

X 9

Commands, control flow

- Do this, then do that
- “X” is the name of a cell that has different values at different times

Imperative

C, C++, Java, C#, VB



```
X := In1  
X := X*X  
X := X + In2*In2
```

In1 3

In2 4

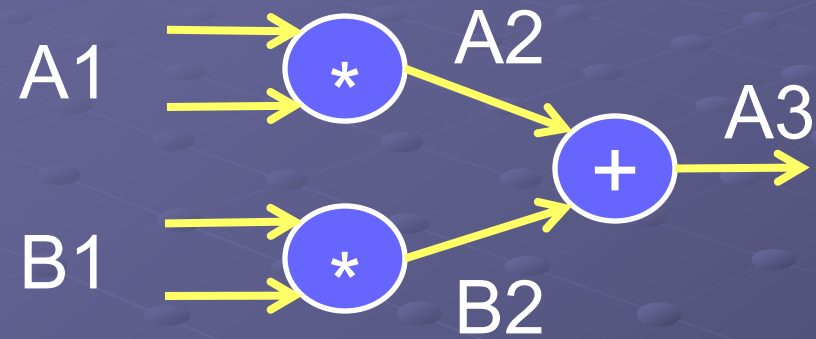
X 25

Commands, control flow

- Do this, then do that
- “X” is the name of a cell that has different values at different times

Functional

Excel, Haskell



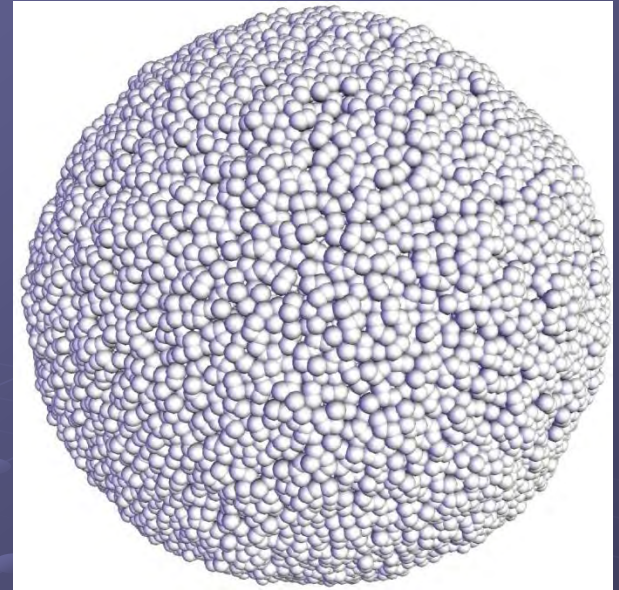
	A	B
1	3	4
2	=A1*A1	=B1*B1
3	=A2+B2	

$A2 = A1 * A1$
 $B2 = B1 * B1$
 $A3 = A2 + B2$

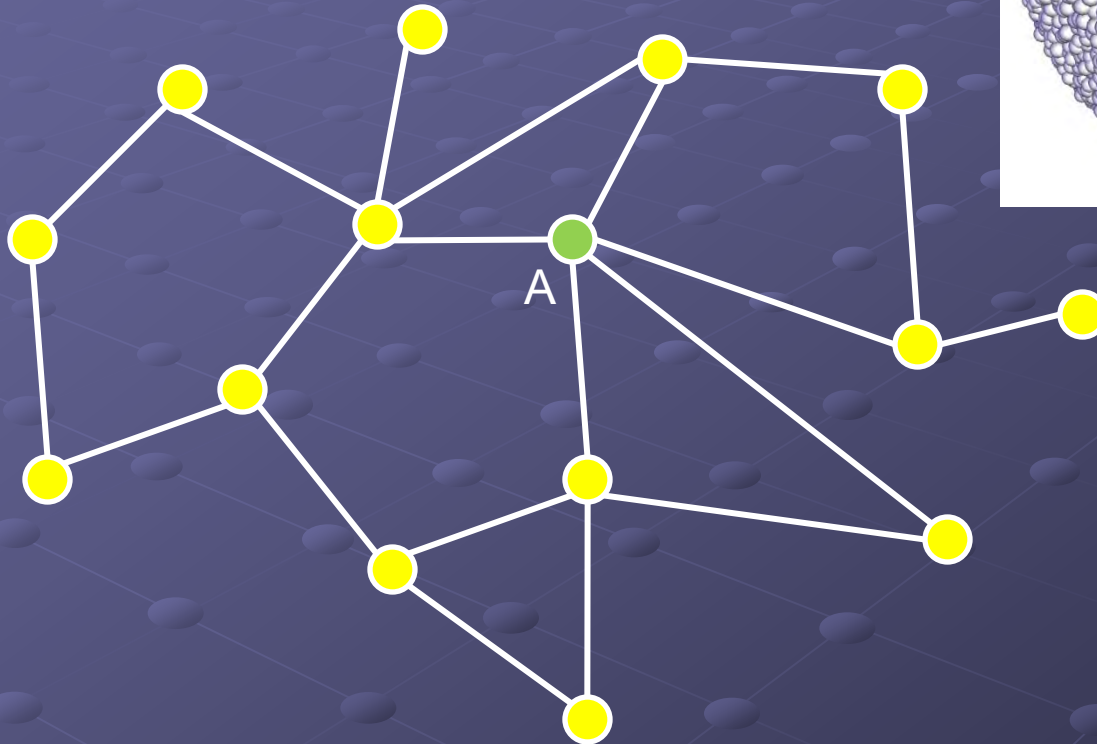
Expressions, data flow

- No notion of sequence
- “A2” is the name of a (single) value

A bigger example



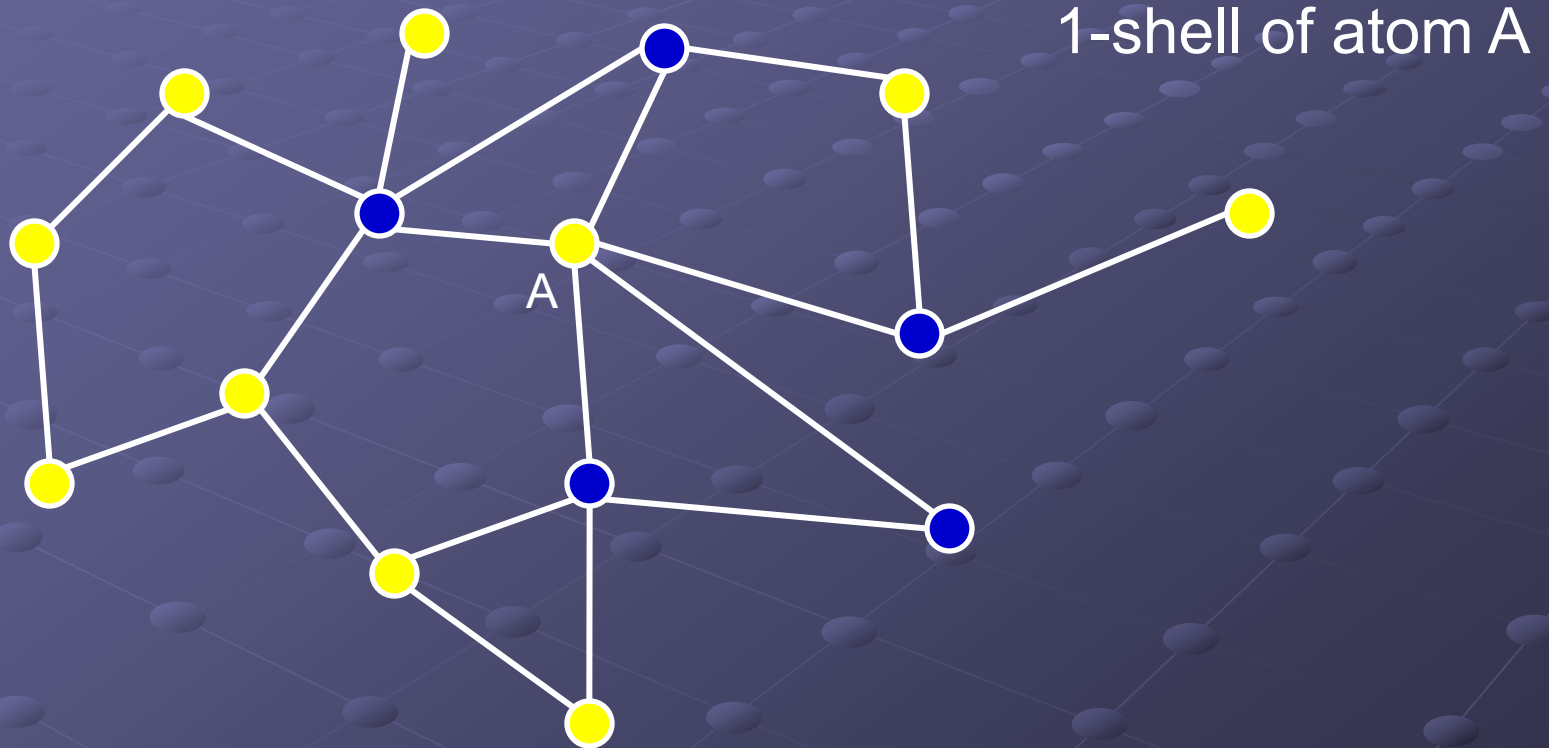
50-shell of 100k-atom model
of amorphous silicon,
generated using F#
Thanks: Jon Harrop



N-shell of atom A

Atoms accessible in N hops (but no fewer) from A

A bigger example

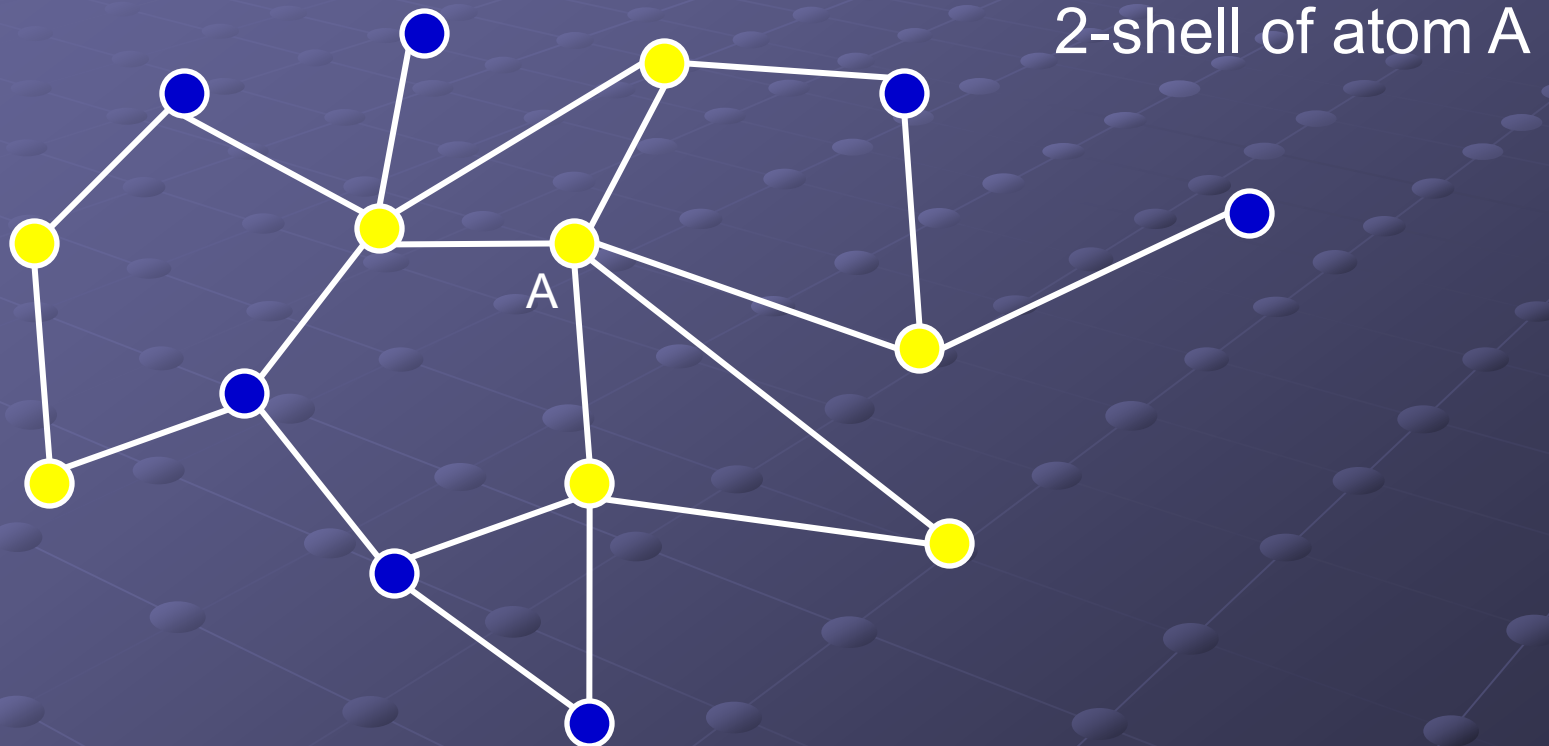


1-shell of atom A

N-shell of atom A

Atoms accessible in N hops (but no fewer) from A

A bigger example

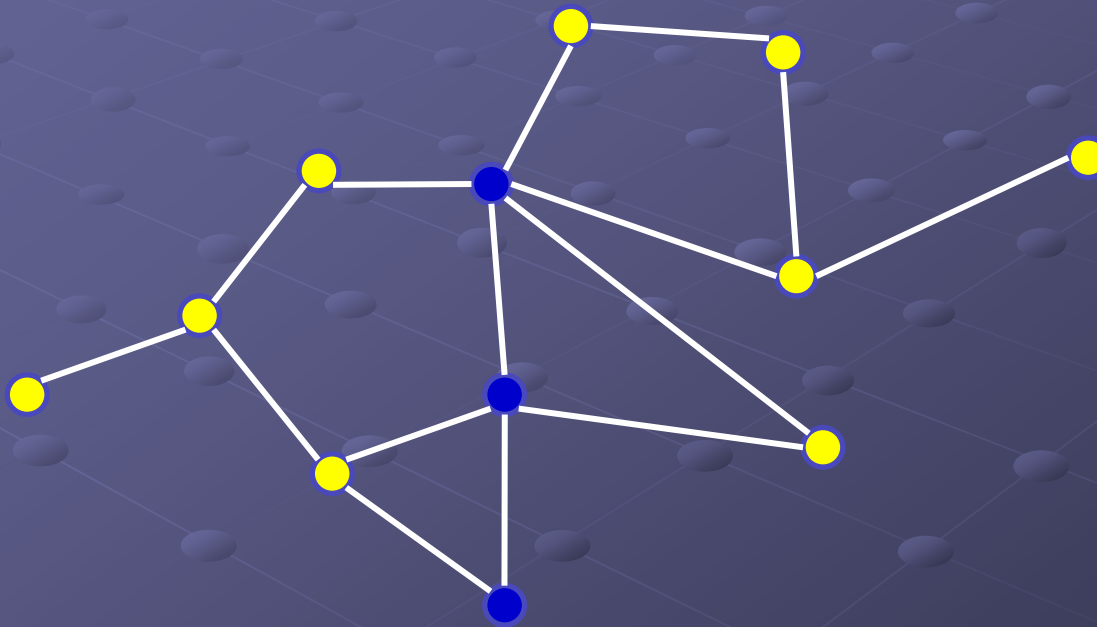


Atoms accessible in N hops (but no fewer) from A

A bigger example

To find the N-shell of A

- Find the (N-1) shell of A
- Union the 1-shells of each of those atoms
- Delete the (N-2) shell and (N-1) shell of A



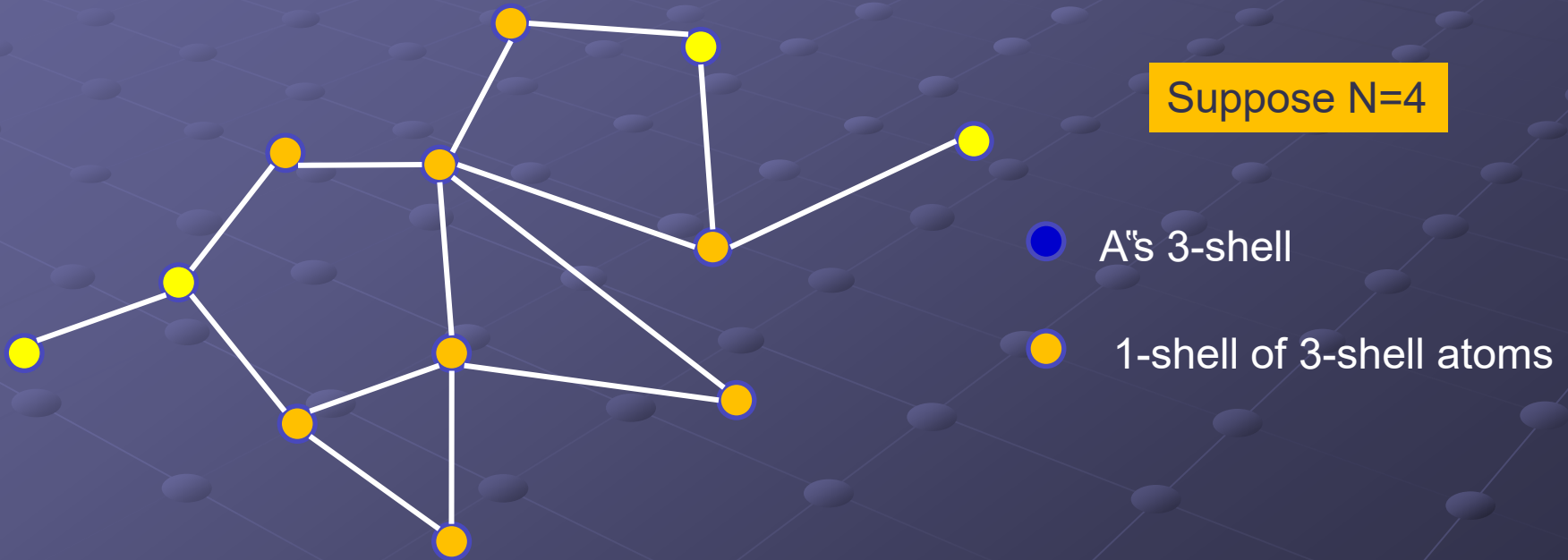
Suppose $N=4$

● A's 3-shell

A bigger example

To find the N-shell of A

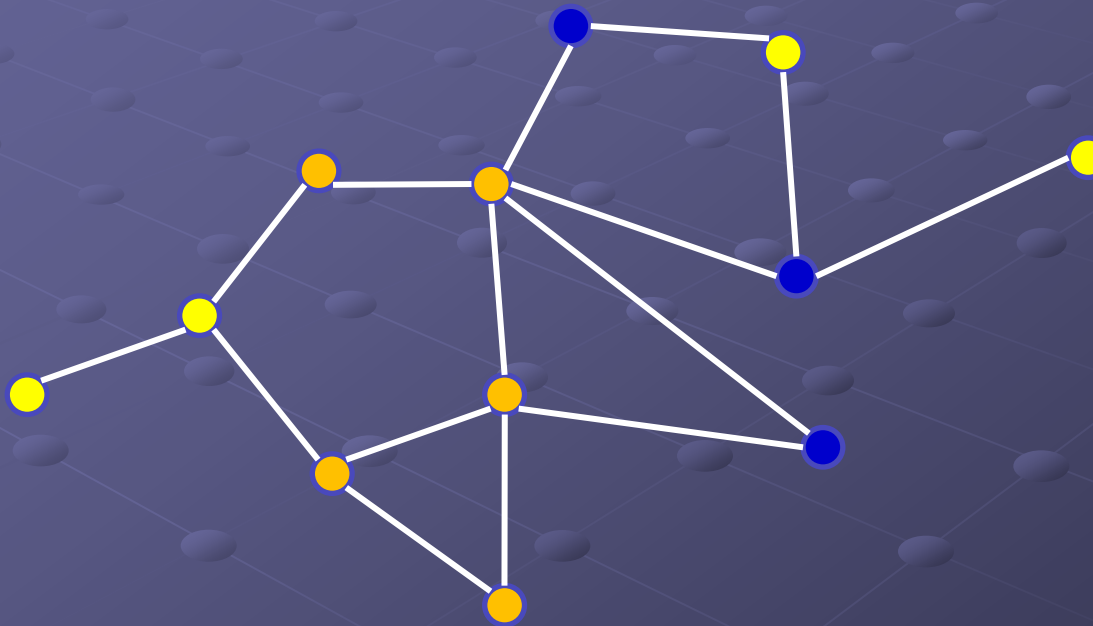
- Find the (N-1) shell of A
- **Union the 1-shells of each of those atoms**
- Delete the (N-2) shell and (N-1) shell of A



A bigger example

To find the N-shell of A

- Find the (N-1) shell of A
- Union the 1-shells of each of those atoms
- **Delete the (N-2) shell and (N-1) shell of A**



Suppose $N=4$

- A's 2-shell and 3-shell
- A's 4-shell

A bigger example

To find the N-shell of A

- Find the (N-1) shell of A
- Find all the neighbours of those atoms
- Delete the (N-2) shell and (N-1) shell of A

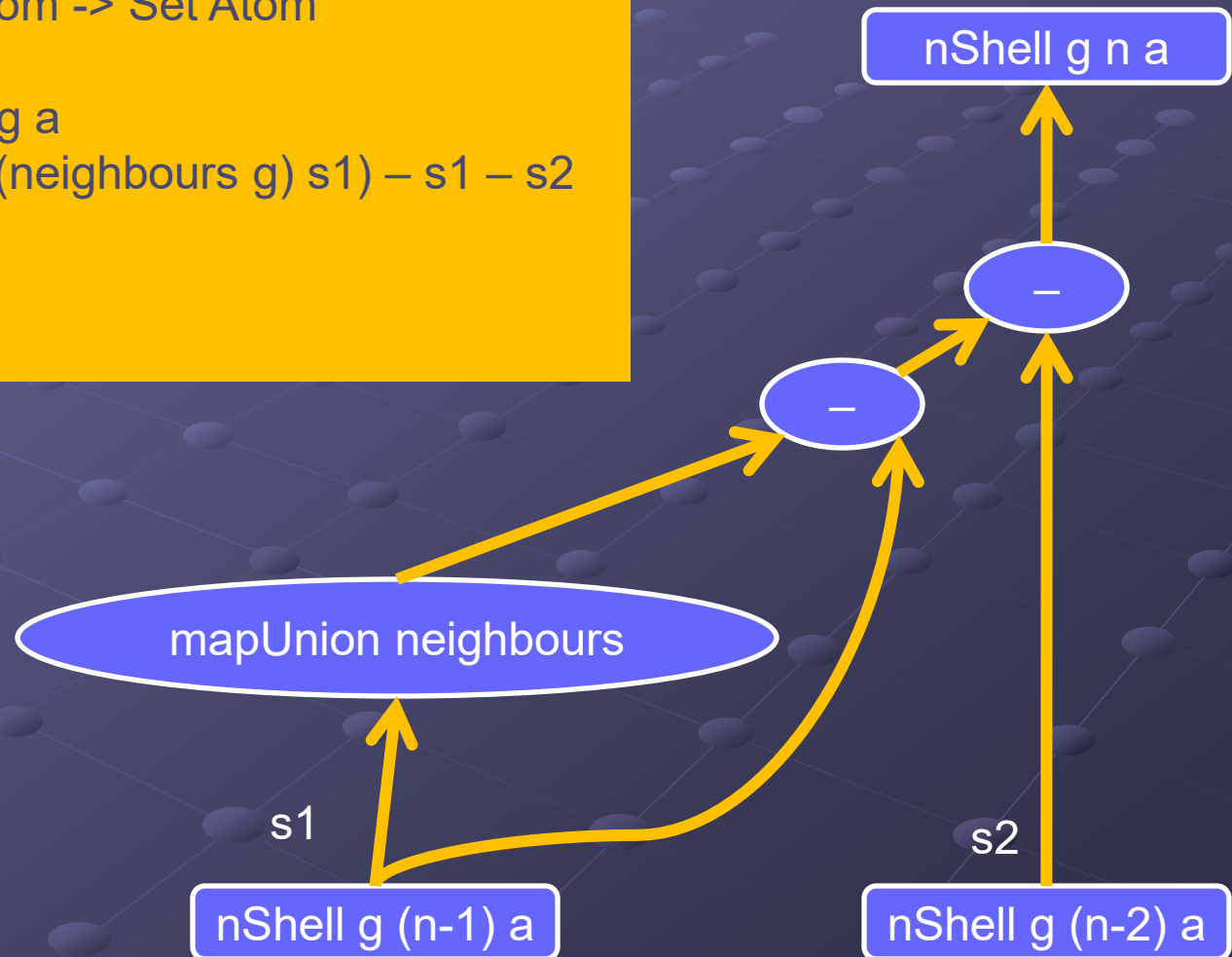
```
(-)           :: Set a -> Set a -> Set a  
mapUnion     :: (a -> Set b) -> Set a -> Set b  
  
neighbours   :: Graph -> Atom -> Set Atom
```

```
nShell :: Graph -> Int -> Atom -> Set Atom  
nShell g 0 a = unitSet a  
nShell g 1 a = neighbours g a  
nShell g n a = (mapUnion (neighbours g) s1) - s1 - s2  
  where  
    s1 = nShell g (n-1) a  
    s2 = nShell g (n-2) a
```

A bigger example

$(-)$:: Set a -> Set a -> Set a
mapUnion :: (a -> Set b) -> Set a -> Set b
neighbours :: Graph -> Atom -> Set Atom

```
nShell :: Graph -> Int -> Atom -> Set Atom
nShell g 0 a = unitSet a
nShell g 1 a = neighbours g a
nShell g n a = (mapUnion (neighbours g) s1) - s1 - s2
  where
    s1 = nShell g (n-1) a
    s2 = nShell g (n-2) a
```



But...

$\text{nShell} :: \text{Graph} \rightarrow \text{Int} \rightarrow \text{Atom} \rightarrow \text{Set Atom}$

$\text{nShell } g \ 0 \ a = \text{unitSet } a$

$\text{nShell } g \ 1 \ a = \text{neighbours } g \ a$

$\text{nShell } g \ n \ a = (\text{mapUnion } (\text{neighbours } g) \ s1) - s1 - s2$

where

$s1 = \text{nShell } g \ (n-1) \ a$

$s2 = \text{nShell } g \ (n-2) \ a$

nShell n needs

- **nShell (n-1)**
- **nShell (n-2)**

But...

```
nShell :: Graph -> Int -> Atom -> Set Atom
nShell g 0 a = unitSet a
nShell g 1 a = neighbours g a
nShell g n a = (mapUnion (neighbours g) s1) - s1 - s2
  where
    s1 = nShell g (n-1) a
    s2 = nShell g (n-2) a
```

nShell n needs

- **nShell (n-1)** which needs
 - **nShell (n-2)**
 - **nShell (n-3)**
- **nShell (n-2)** which needs
 - **nShell (n-3)**
 - **nShell (n-4)**

Duplicates!

But...

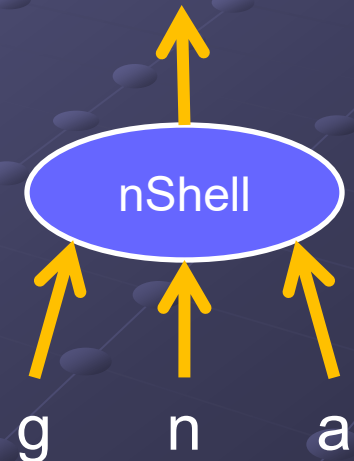
```
nShell :: Graph -> Int -> Atom -> Set Atom
nShell g 0 a = unitSet a
nShell g 1 a = neighbours g a
nShell g n a = (mapUnion (neighbours g) s1) - s1 - s2
  where
    s1 = nShell g (n-1) a
    s2 = nShell g (n-2) a
```

BUT, the two calls to $(nShell\ g\ (n-2)\ a)$
must yield the same result

And so we can safely share them

- Memo function, or
- Return a pair of results

**Same inputs
means
same outputs**



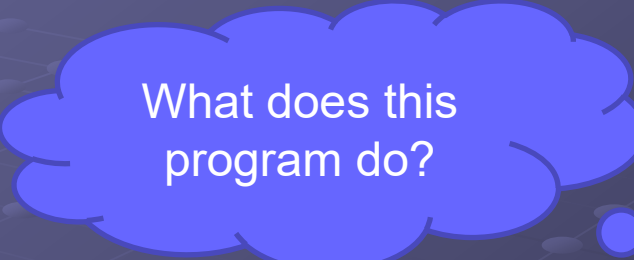
“Purity”

“Referential transparency”

“No side effects”

Purity pays: understanding

```
X1.insert( Y )  
X2.delete( Y )
```



What does this
program do?

- Would it matter if we swapped the order of these two calls?
- What if $X1=X2$?
- I wonder what **else** X1.insert does?

Lots of heroic work on static analysis, but hampered by unnecessary effects

Purity pays: verification

Pre-condition

Spec#

```
void Insert( int index, object value )  
  requires (0 <= index && index <= Count)  
  ensures Forall{ int i in 0:index; old(this[i]) == this[i] }  
  { ... }
```

Post-condition

- The pre and post-conditions are written in... a functional language
- Also: object invariants
But: invariants temporarily broken
Hence: “expose” statements

Purity pays: testing

A property of sets

$$s \cup s = s$$

```
propUnion :: Set a -> Bool
```

```
propUnion s = union s s == s
```

In an imperative or OO language, you must

- set up the state of the object, and the external state it reads or writes
- make the call
- inspect the state of the object, and the external state
- perhaps copy part of the object or global state, so that you can use it in the postcondition

Purity pays: maintenance

- The **type** of a function tells you a LOT about it

```
reverse :: [a] -> [a]
```

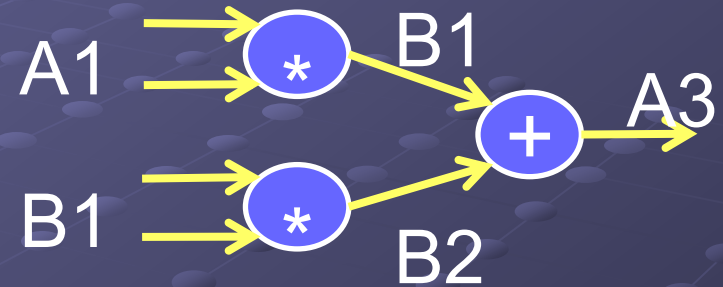
- Large-scale data representation changes in a multi-100kloc code base can be done reliably:
 - change the representation
 - compile until no type errors
 - works

Purity pays: performance

- Execution model is not so close to machine
 - Hence, bigger job for compiler, execution may be slower
- But: algorithm is often more important than raw efficiency
- And: purity supports radical optimisations
 - nShell runs 100x faster in F# than C++
Why? More sharing of parts of sets.
 - SQL, XQuery query optimisers
- Real-life example: Smoke Vector Graphics library: 200kloc C++ became 50kloc OCaml, and ran 5x faster

Purity pays: parallelism

- Pure programs are “naturally parallel”
- No mutable state means no locks, no race hazards
- Results totally unaffected by parallelism (1 processor or zillions)
- Examples
 - Google’s map/reduce
 - SQL on clusters
 - PLINQ



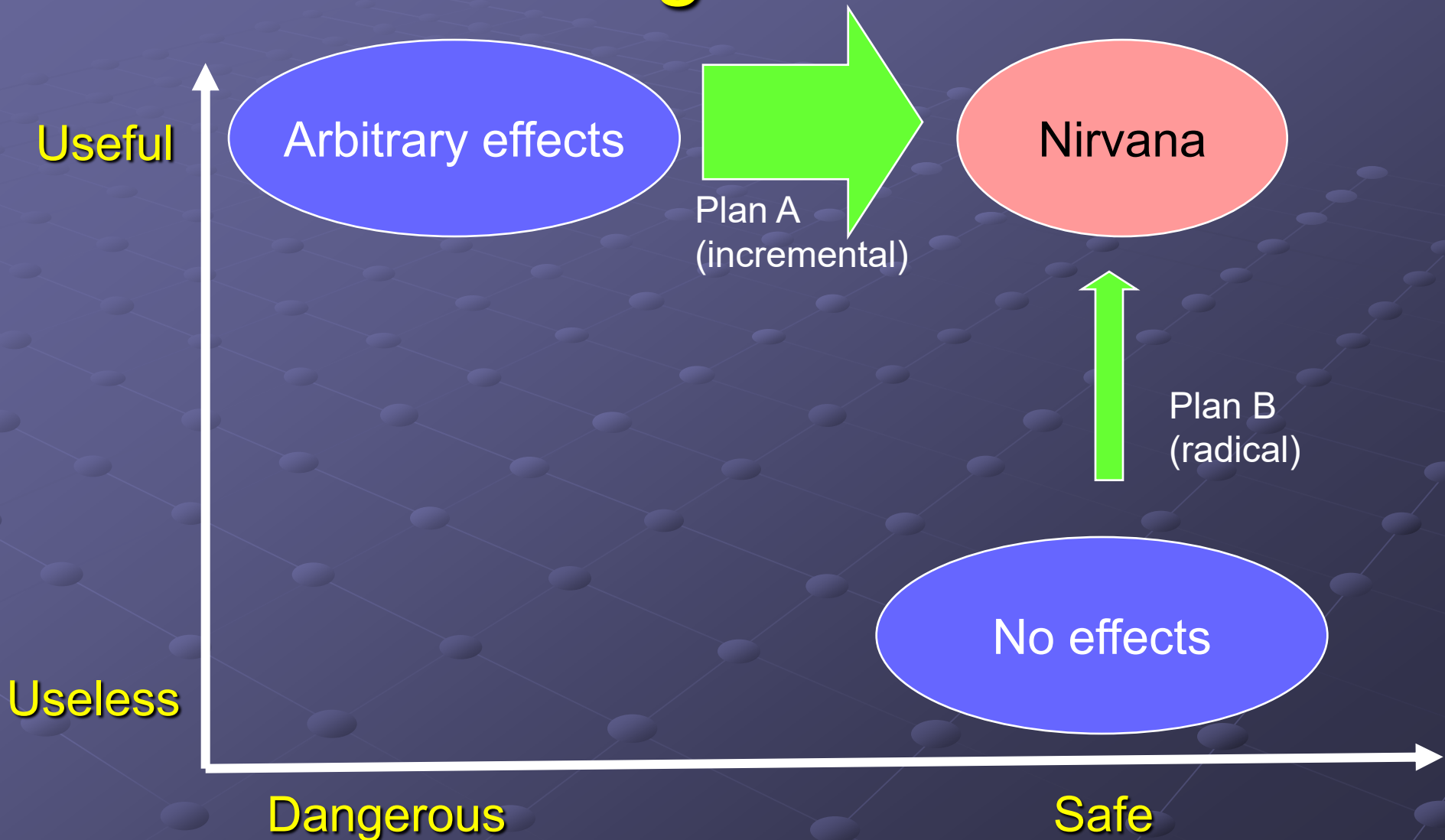
Purity pays: parallelism

Can I run this LINQ query in parallel?

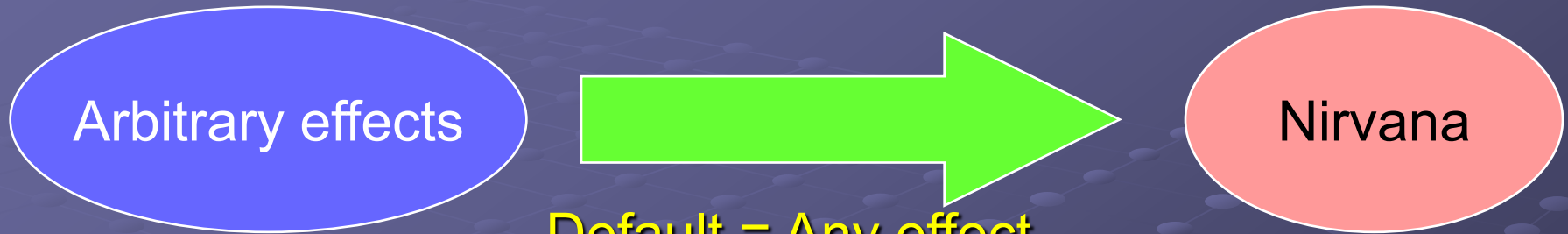
```
int index = 0;  
List<Customer> top10 = (from c in customers  
                        where index++ < 10  
                        select c).ToList();
```

- Race hazard because of the side effect in the „where“ clause
- May be concealed inside calls
- Parallel query is correct/reliable only if the expressions in the query are 100% pure

The central challenge: taming effects



Plan A: build on what we have

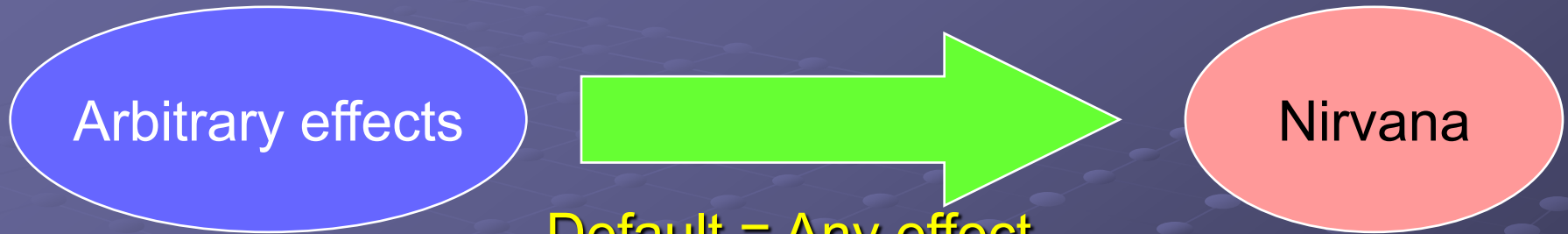


Default = Any effect
Plan = Add restrictions

Erlang

- No mutable variables
- Limited effects
 - send/receive messages,
 - input/output,
 - exceptions
- Rich pure sub-language: lists, tuples, higher order functions, comprehensions, pattern matching...

Plan A: build on what we have

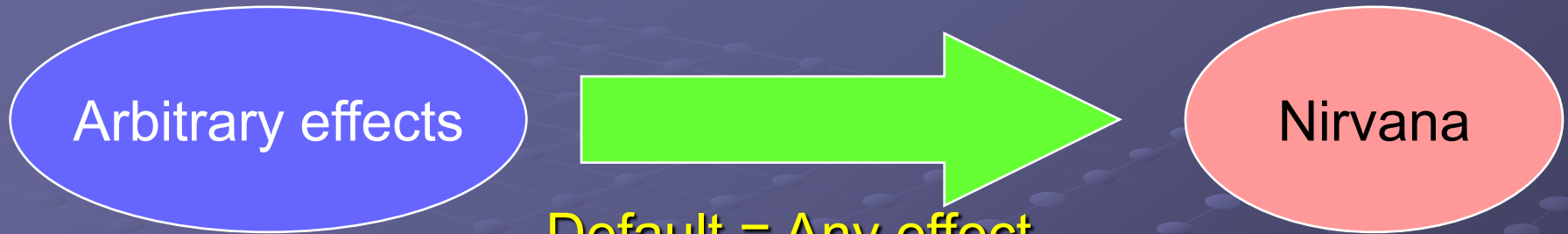


Default = Any effect
Plan = Add restrictions

F#

- A .NET language; hence unlimited effects
- But, a rich pure sub-language: lists, tuples, higher order functions, comprehensions, pattern matching...

Plan A: build on what we have



Default = Any effect
Plan = Add restrictions

BUT

**How do we know
(for sure) that a
function is pure?**

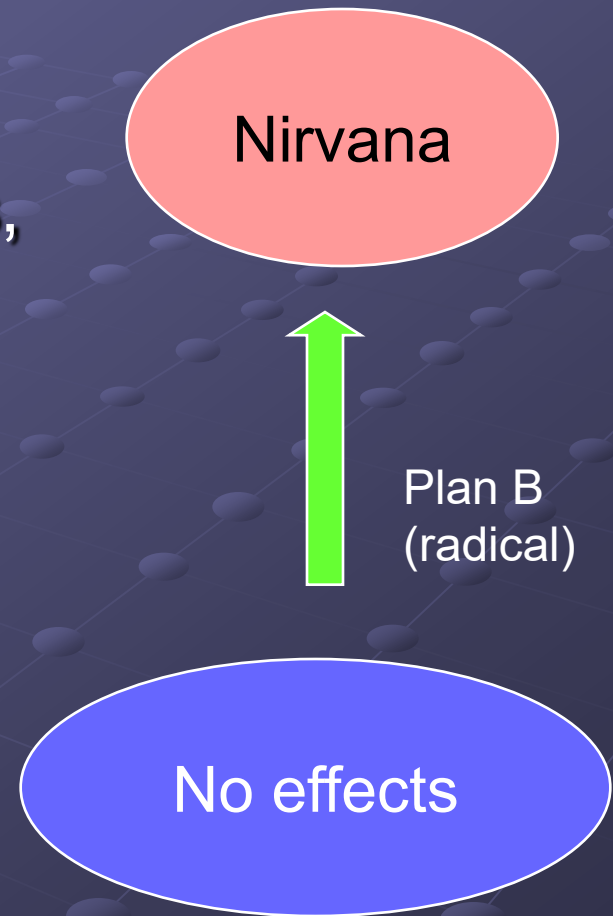
Plan A answer: by convention

Plan B: purity by default

Haskell

- A rich pure language: lists, tuples, higher order functions, comprehensions, pattern matching...
- NO side effects at all

Hmm... ultimately, the program must have SOME effect!



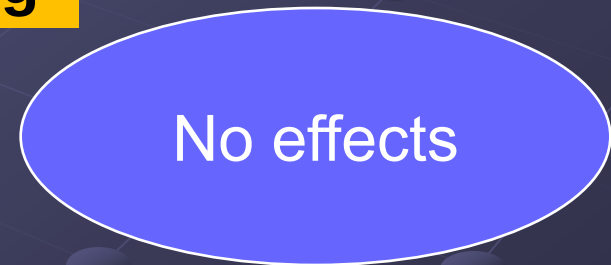
Plan B: purity by default

Haskell

- We learned how to do I/O using so-called “monads”
- Pure function:
`toUpper :: String -> String`
- Side-effecting function
`getUserInput :: String -> IO String`
- The type tells (nearly) all



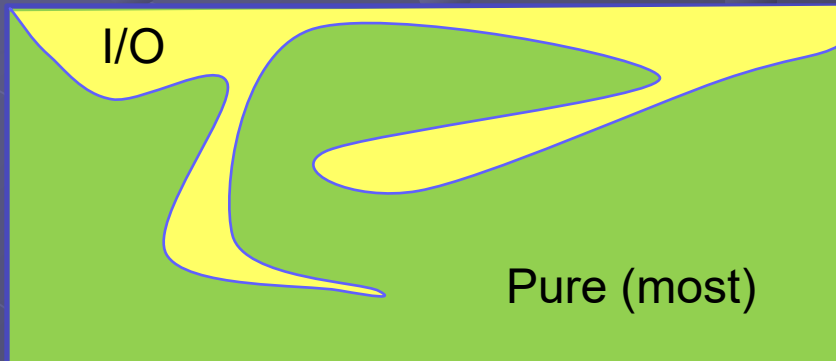
Plan B
(radical)



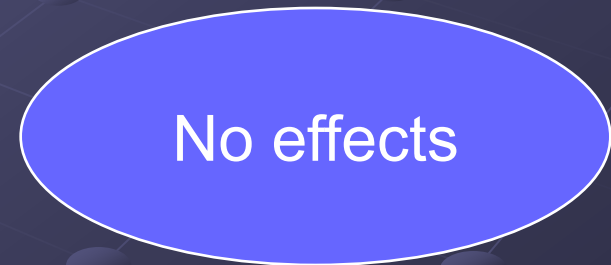
Plan B: purity by default

Haskell

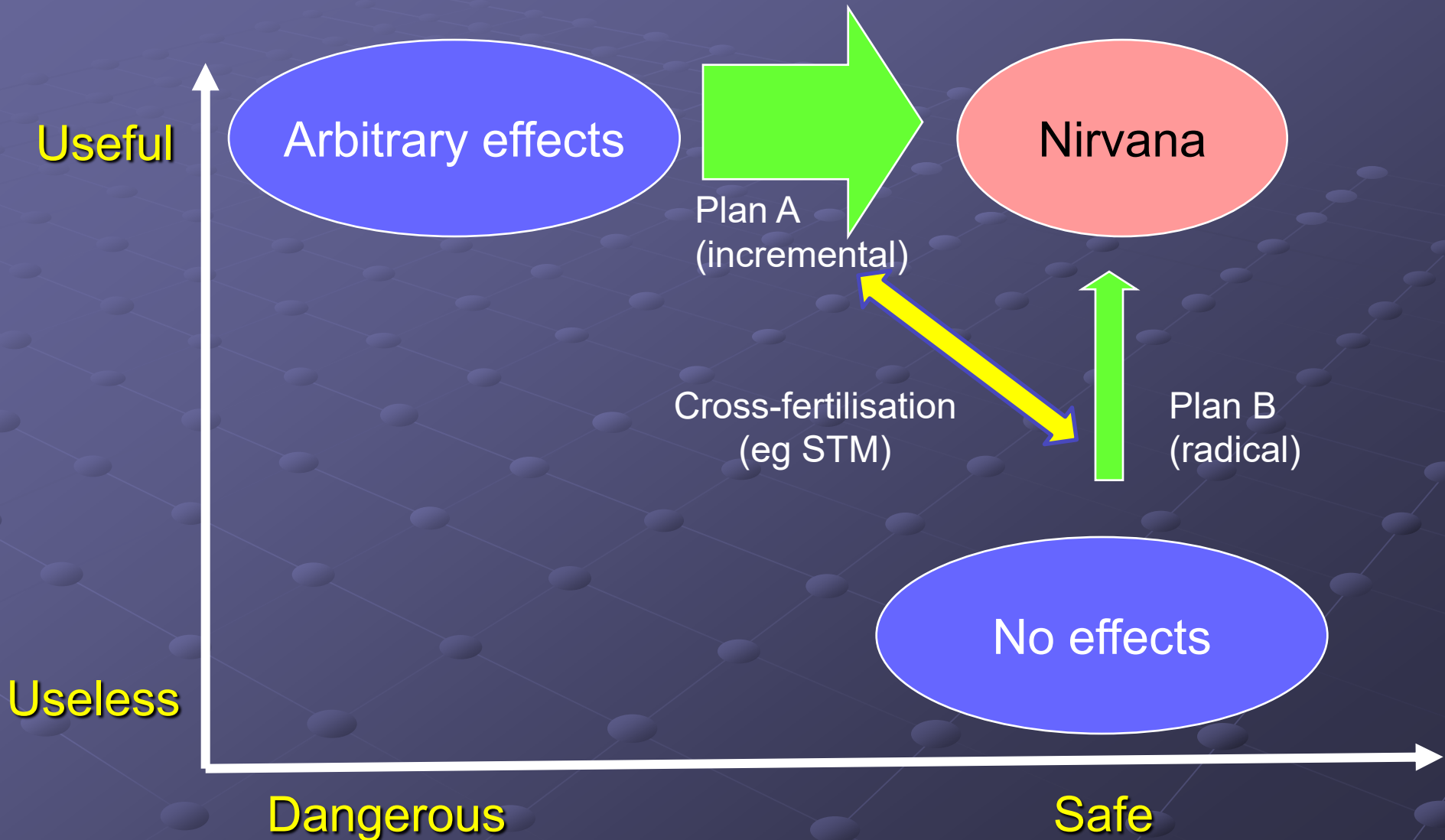
- The type tells (nearly) all
- A single program is a mixture of pure and effect-ful code, kept hermetically separated by the type system



Plan B
(radical)



The central challenge



Effects matter: transactions

- Multiple threads with shared, mutable state
- Brand leader: locks and condition variables
- New kid on the block: transactional memory

```
atomic    { withdraw( A, 4 )  
            ; deposit (B, 4 ) }
```

- Optimistic concurrency:
 - run code without taking locks, logging changes
 - check at end whether transaction has seen a consistent view of memory
 - if so, commit effects to shared memory
 - if not, abort and re-run transaction

Effects matter: transactions

- TM only make sense if the transacted code
 - Does no input output
 - Mutates only transacted variables
- So effects form a **spectrum**



- Monads classify the effects

```
transferMoney :: Acc -> Acc -> Int -> STM ()
```

```
getUserInput :: String -> IO String
```

Can do
arbitrary I/O

Can only
read/write Tvars
No I/O!

My claims

- Mainstream languages are hamstrung by gratuitous (ie unnecessary) effects

```
T = 0; for (i=0; i<N; i++) { T = T + i }
```

Effects are part of the fabric of computation

- Future software will be effect-free by default,
 - With controlled effects where necessary
 - Statically checked by the type system

And the future is here...

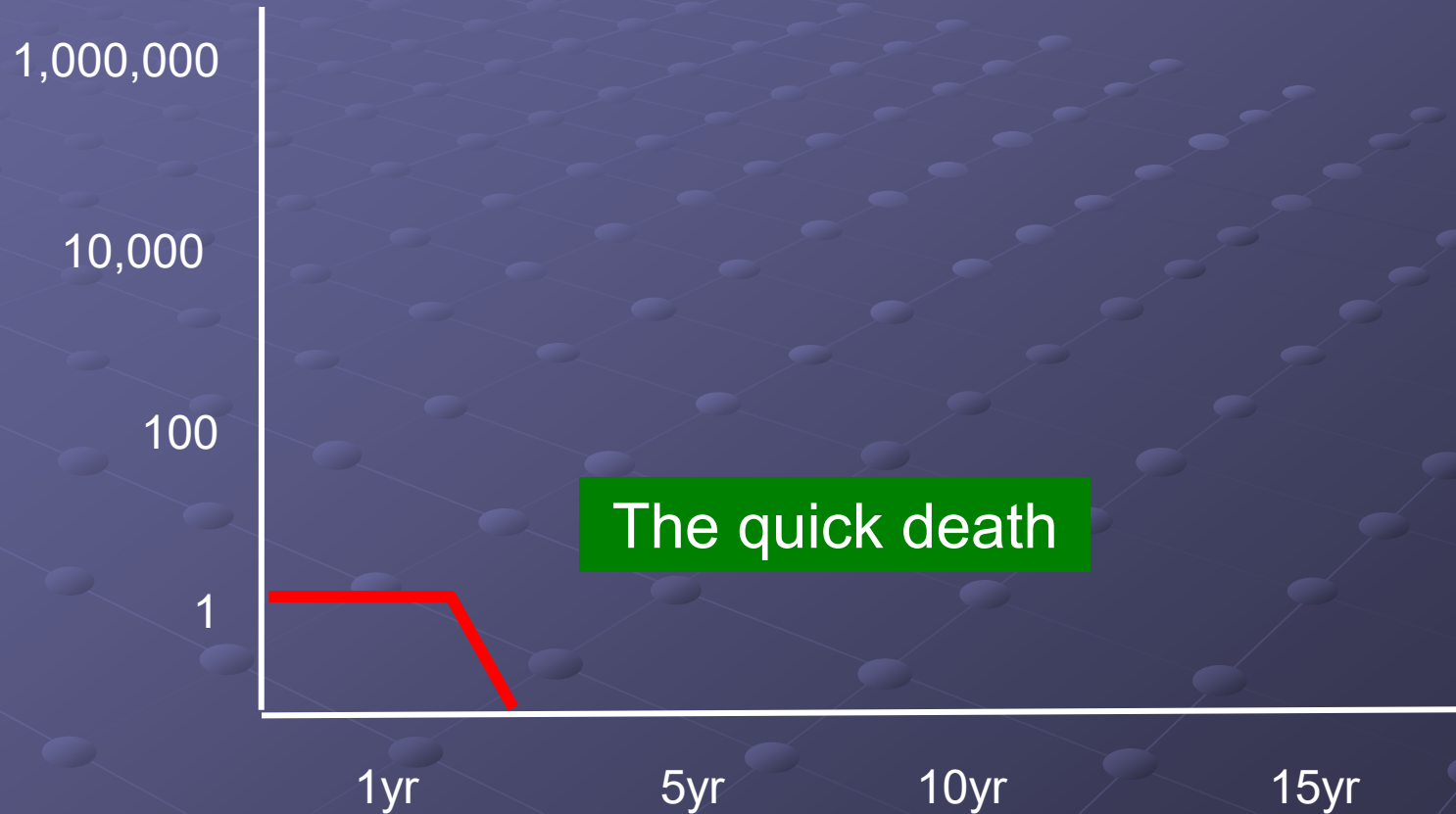
- Functional programming has fascinated academics for decades
- But professional-developer interest in functional programming has sky-rocketed in the last 5 years.

Suddenly, FP is cool, not geeky.

Most research languages

Practitioners

Geeks



The quick death

Successful research languages

Practitioners

Geeks



C++, Java, Perl, Ruby

Practitioners

Geeks

1,000,000

10,000

100

1

1yr

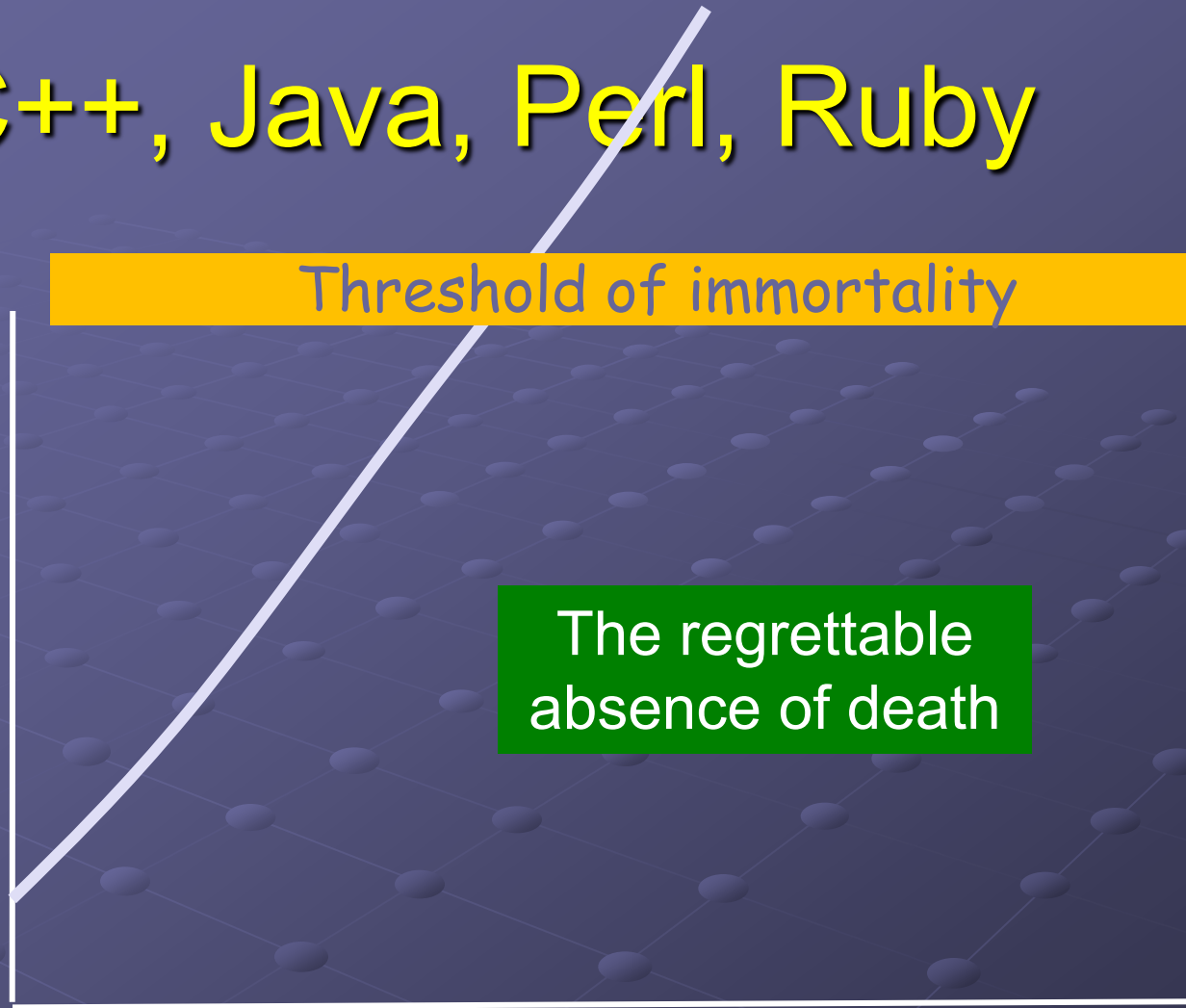
5yr

10yr

15yr

Threshold of immortality

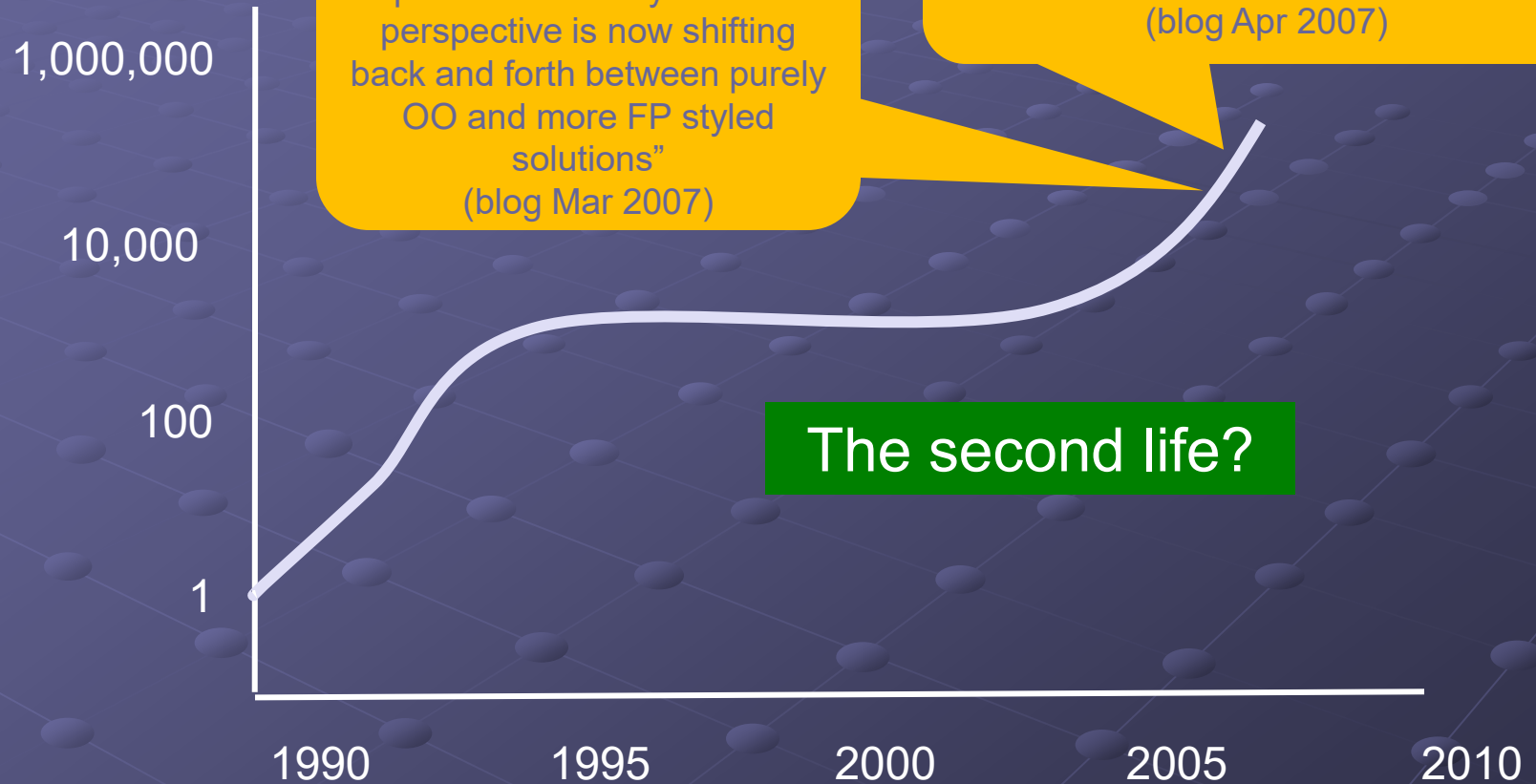
The regrettable
absence of death



Haskell

Practitioners

Geeks

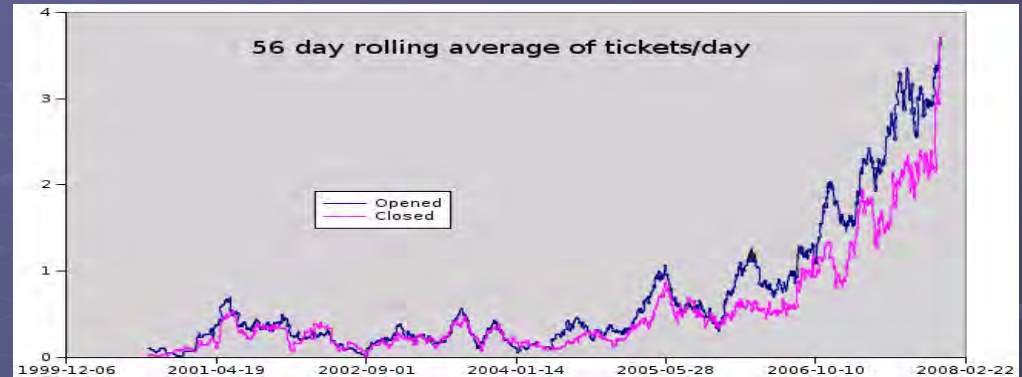


Lots of other great examples

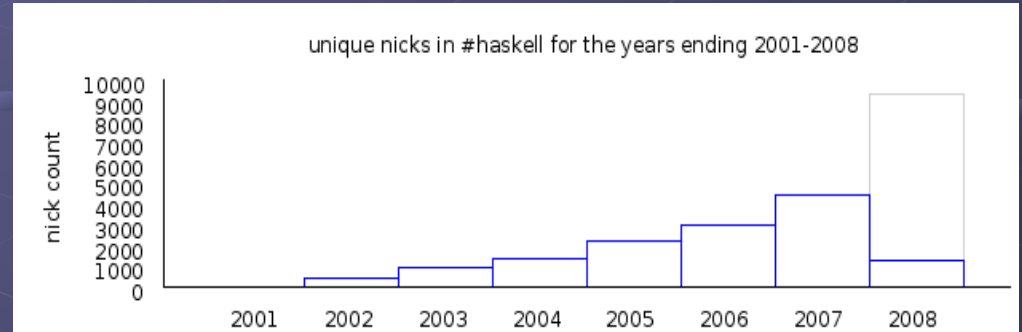
- **Erlang**: widely respected and admired as a shining example of functional programming applied to an important domain
- **F#**: now being commercialised by Microsoft
- **OCaml, Scala, Scheme**: academic languages being widely used in industry
- **C#**: explicitly adopting functional ideas (e.g. LINQ)

Sharply rising activity

GHC bug tracker
1999-2007



Haskell IRC channel
2001-2007



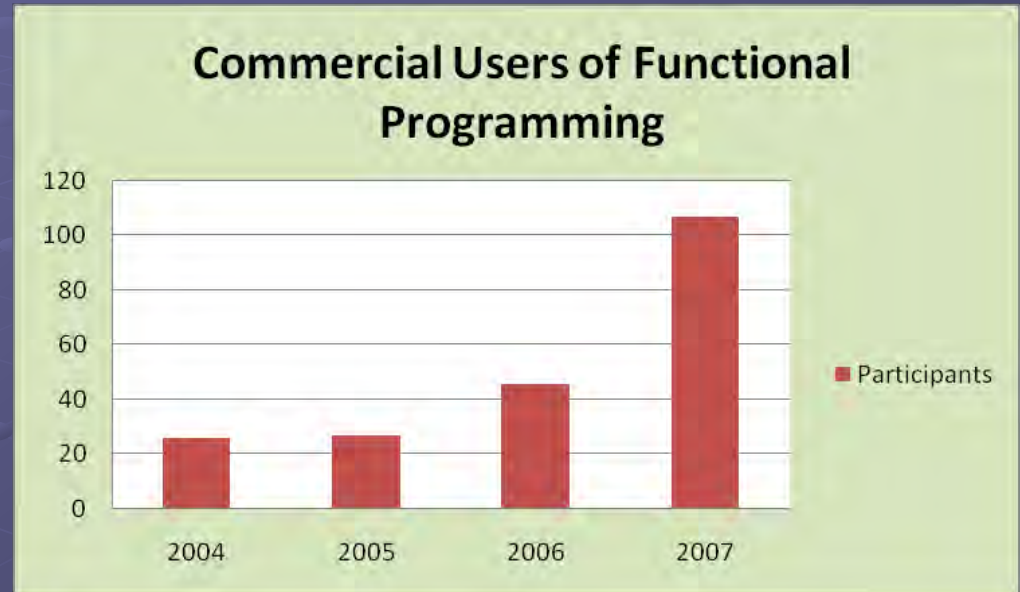
Jan 20	Austin Functional Programming
Feb 9	FringeDC
Feb 11	PDXFunc
Feb 12	Fun in the afternoon
Feb 13	BayFP
Feb 16	St-Petersburg Haskell User Group
Feb 19	NYFP Network
Feb 20	Seattle FP Group

Austin
Washington DC
Portland
London
San Francisco
Saint-Petersburg
New York
Seattle

CUFP

Commercial Users of Functional Programming 2004-2007

Speakers describing applications in:
banking, smart cards, telecoms, data
parallel, terrorism response training,
machine learning, network services,
hardware design, communications
security, cross-domain security



CUFP 2008 is part of the a new

Functional Programming Developer Conference

(tutorials, tools, recruitment, etc)

Victoria, British Columbia, Sept 2008

Same meeting: workshops on Erlang, ML, Haskell, Scheme.

Summary

- The **languages and tools** of functional programming are being used to make money fast
- The **ideas** of functional programming are rapidly becoming mainstream
- In particular, the Big Deal for programming in the next decade is the **control of effects**, and functional programming is the place to look for solutions.

Quotes from the front line

- “Learning Haskell has completely reversed my feeling that static typing is an old outdated idea.”
- “Changing the type of a function in Python will lead to strange runtime errors. But when I modify a Haskell program, I already know it will work once it compiles.”
- “Our chat system was implemented by 3 other groups (two Java, one C++). Haskell implementation is more stable, provides more features, and has about 70% less code.”
- “I’m no expert, but I got an order of magnitude improvement in code size and 2 orders of magnitude development improvement in development time”
- “My Python solution was 50 lines. My Haskell solution was 14 lines, and I was quite pleased. Your Haskell solution was 5.”
- “C isn't hard; programming in C is hard. On the other hand, Haskell is hard, but programming in Haskell is easy.”