Tangible functional programming:

a modern marriage of usability and composability

Conal Elliott

November, 2007

Software applications and libraries have different intentions and strengths.

- user-friendly
  - usable
  - concrete
  - visual
- programmer-friendly
  - composable
  - abstract
  - syntactic

This split has drawbacks.

- Applications limit access to functionality.
- Applications are usually not composable.
- Libraries aren't user-friendly,

  and thus limit access.
- Medium influences & selects message.

So what's the dream?

- Unlimited access to functionality,
  - *usably* and
  - *composably*

Where have we seen

human interface and composition?

Where have we seen

human interface and composition?

Hint:

```
who | sort | lpr
```

“This is the Unix philosophy:

Write programs that do *one thing*
and do it well.

Write programs to *work together*.

”

Doug McIlroy (inventor of Unix pipes)

“This is the Unix philosophy:

Write programs that do *one thing*
and do it well.

Write programs to *work together*.

Write programs to handle *text streams*,
because that is a universal interface.”

Doug McIlroy (inventor of Unix pipes)

If composability means text stream filters, where's my GUI?

"Despite popular mythology, this practice is favored not because Unix programmers hate graphical user interfaces. It's because if you don't write programs that accept and emit simple text streams, *it's much more difficult to hook the programs together*."

Eric Steven Raymond

The Art of Unix Programming

So let’s make it easy.

How can GUI apps compose?

- Translate the pipe idea:
  - Feed output of one piece to input of next.
  - Hide the intermediate data.
- Add: *loosely couple interface & content*.

We'll make some other improvements along the way.

- Graphics
- Data types: convenience & safety
- Multiple arguments (`diff`)
- Scalability/consistency

  (cf. stdio, argv, sockets, threads, C, sh)

Programming is a way to express interfaces and functionality.

- Code is a command-line UI.
- Handy & inessential
- Necessarily indirect

Authoring tools are

functional programming environments.

- In disguise
- Full of interpreted graphs
- Lacks parameterization, type system
- Scripting bolted on

Authoring tools are

functional programming environments.

"Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."

Greenspun's Tenth Rule

Functional programming is a simple and general framework.

- *Value-oriented* programming
- Expressions only – no statements
- I'll address core types of values: atomic (numbers, images, ...), pairs, functions.

Where are we going?

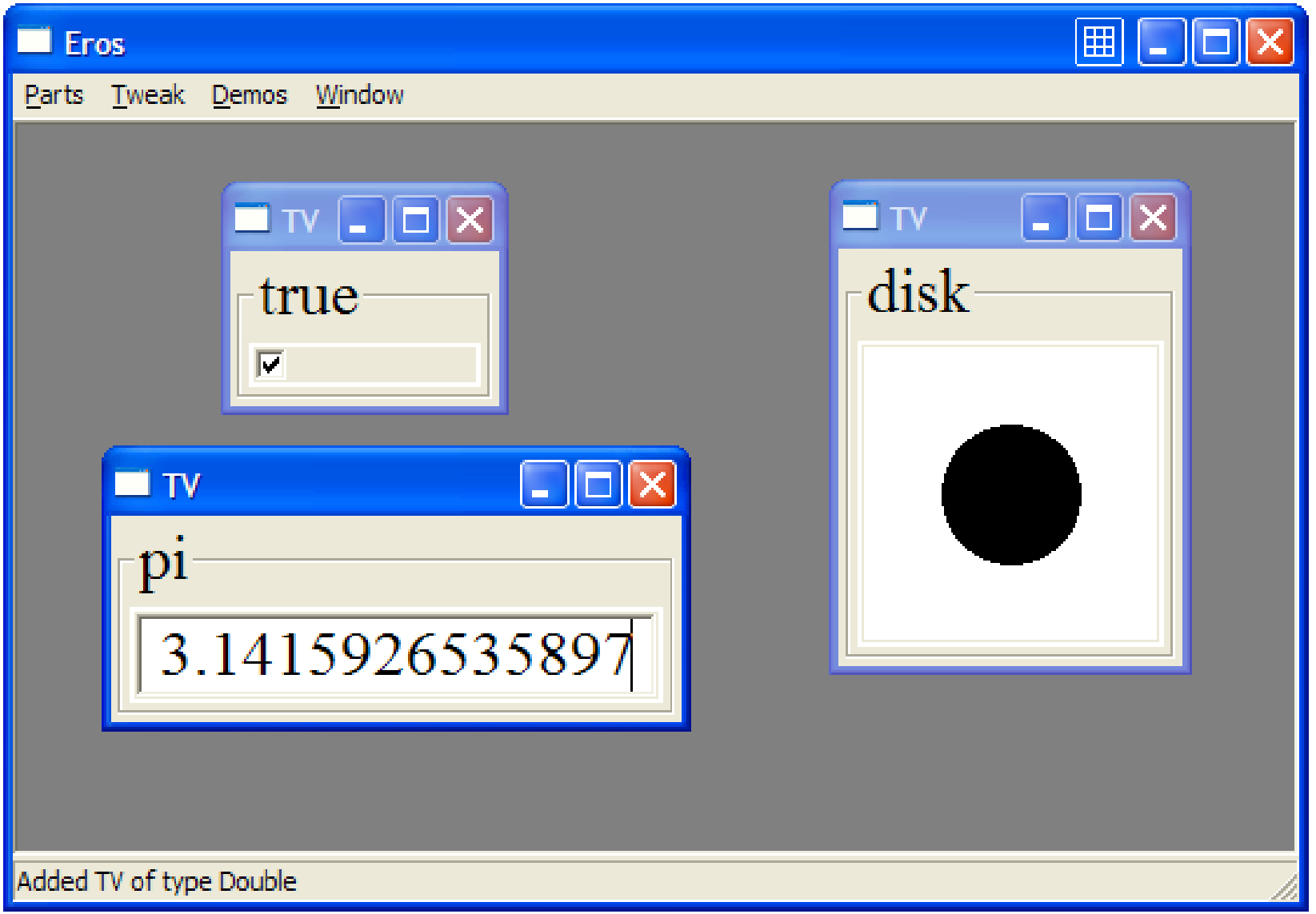
- Usability and composability
- **Eros user experience**
- Mechanics

Key idea #1 (of 4)**:**

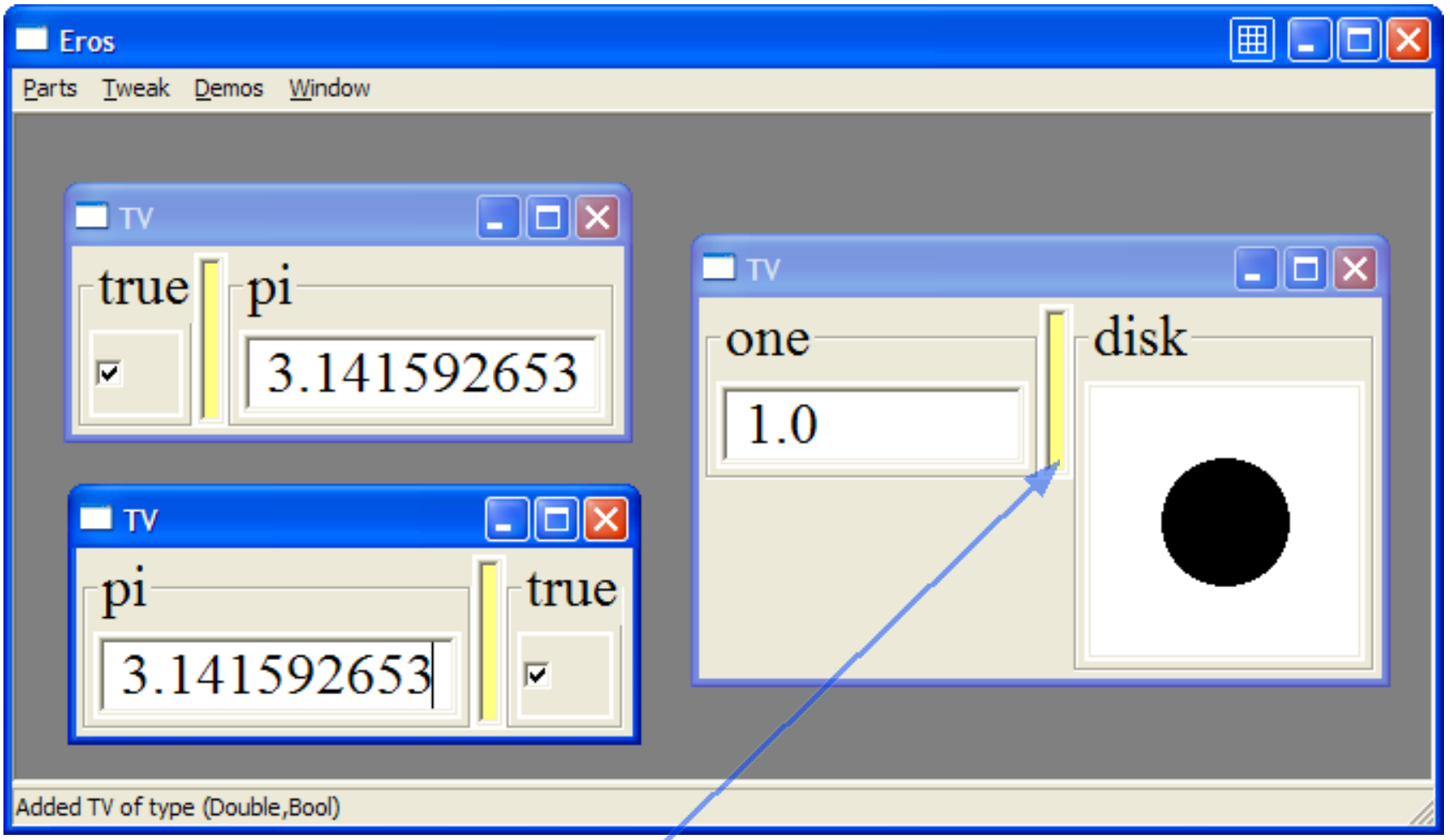
Use GUIs to visualize typed values.

- GUI *structure* follows type.
- GUI *content* presents value.
- Functions visualize as *interactive* GUIs.
- “Tangible values”

Base type values are widgets.



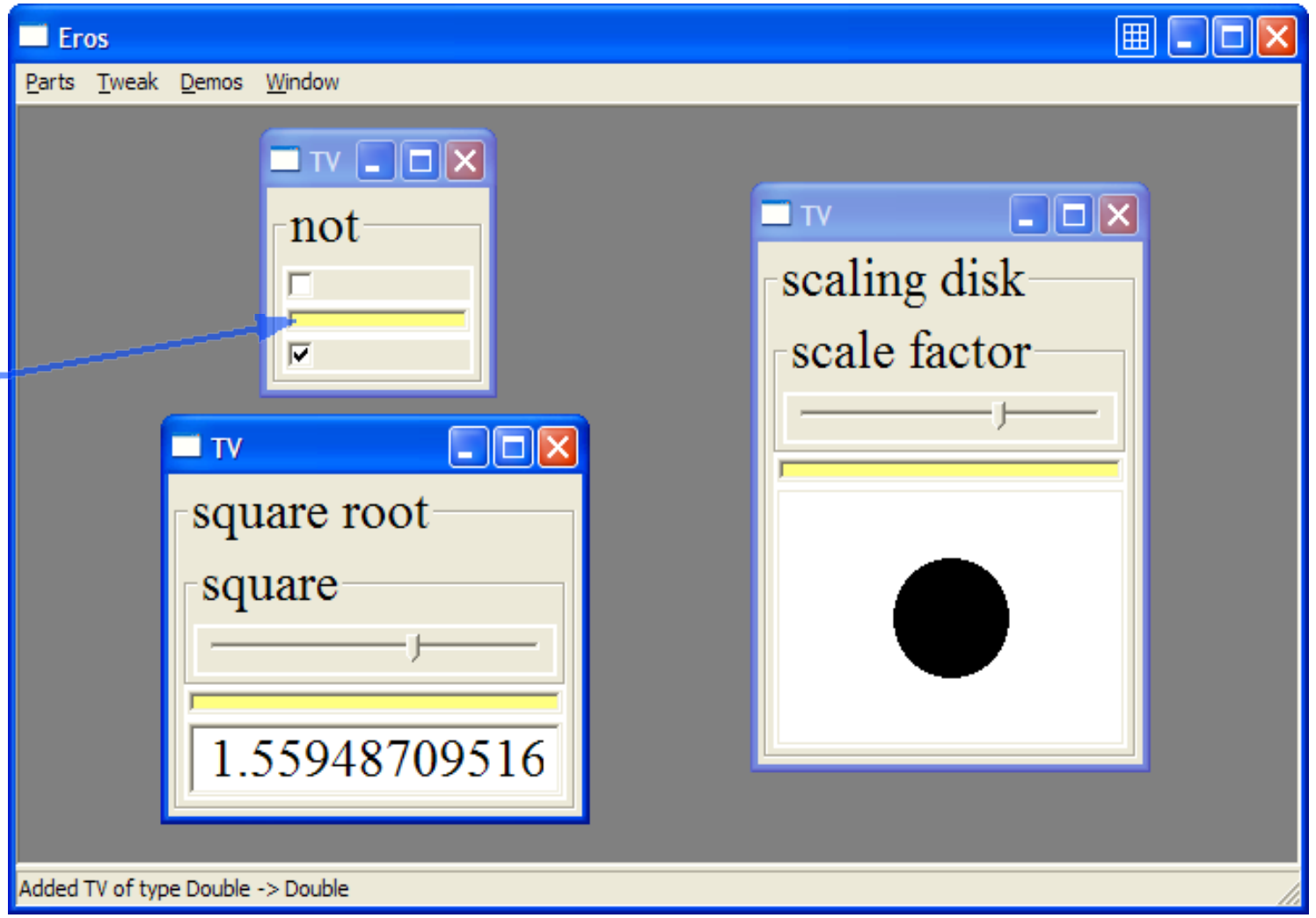
Pairs lay out horizontally.



“,” in (α , β) and (a,b)

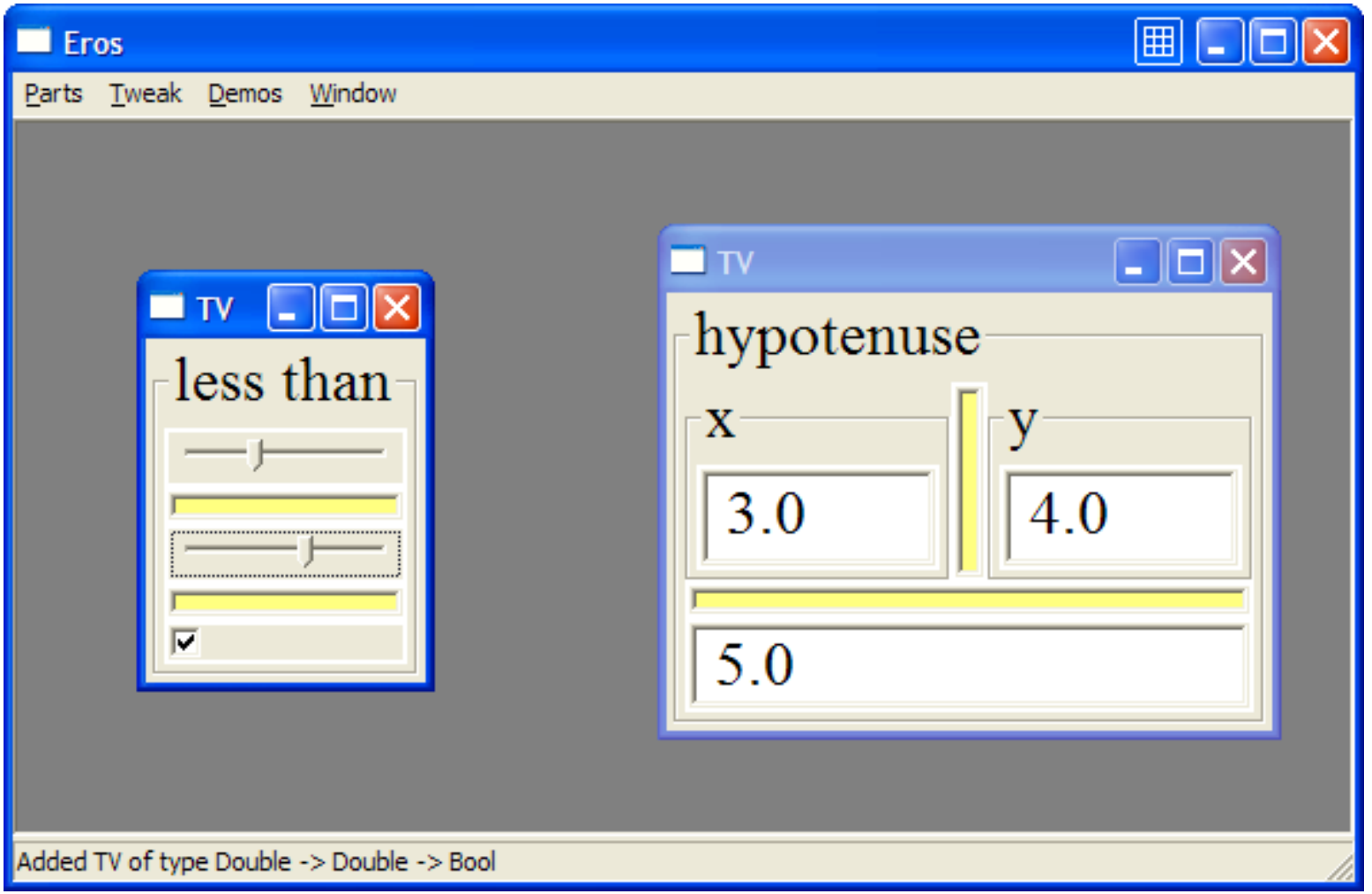
Functions lay out vertically.

“$\rightarrow$”
in
$\alpha \rightarrow \beta$
and
$\lambda\, a \rightarrow b$

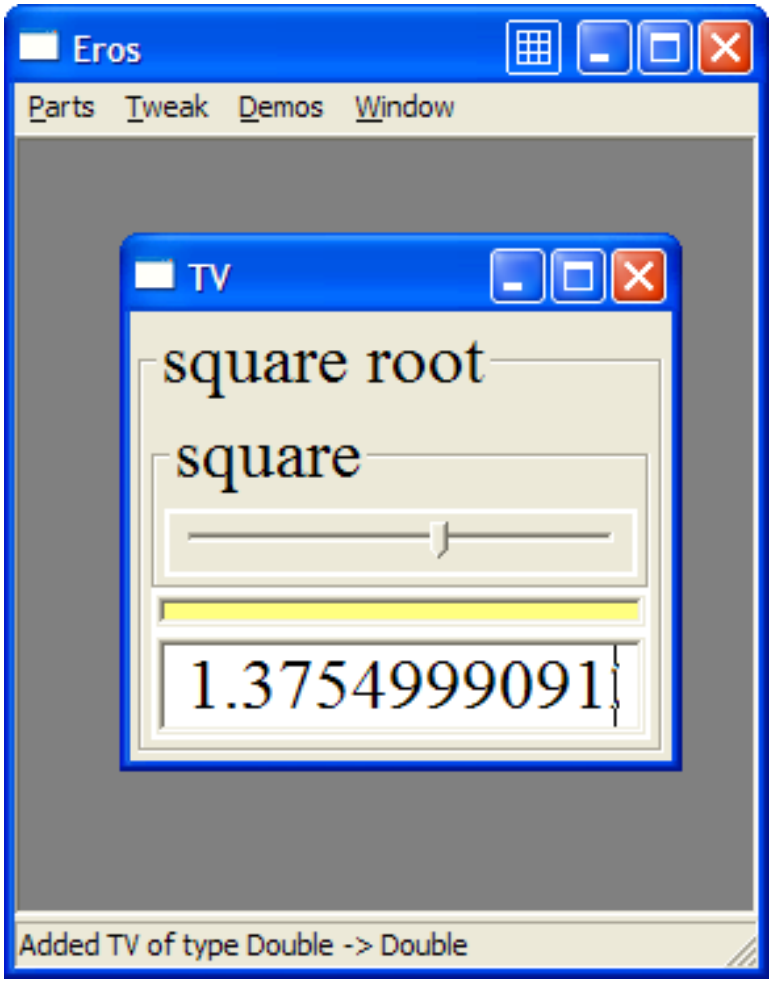


Functions may be

curried or uncurried.



Functions visualize as *interactive* GUIs.

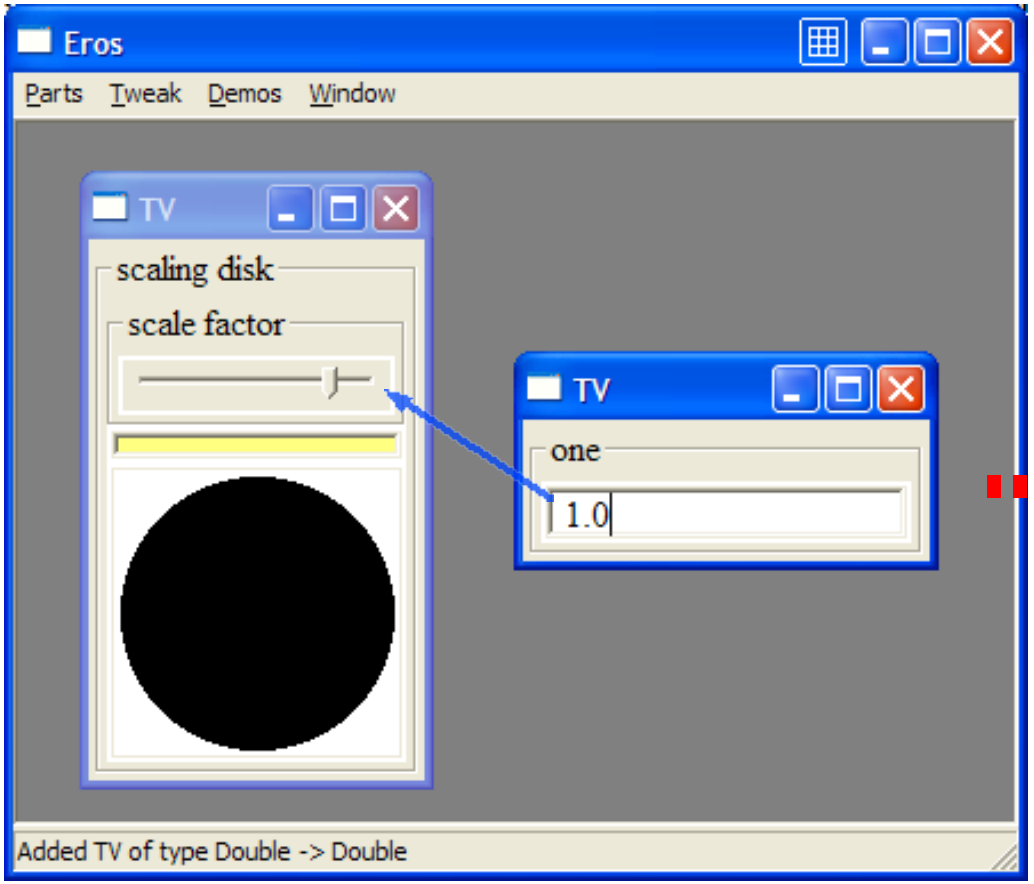


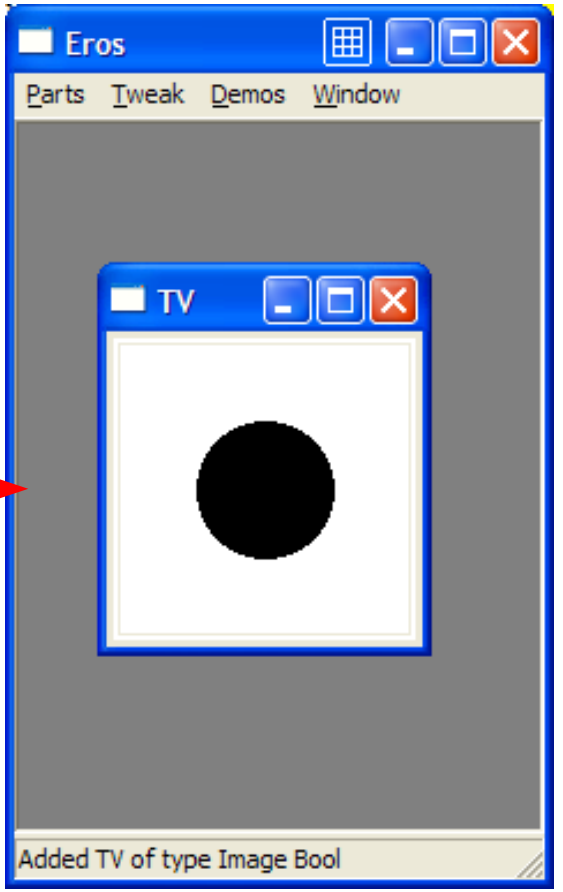
Key idea #2:

Users make new TVs by *fusion*.

- Select compatible input & output,
- which disappear.
- Everything else remains,
- fused into a single new TV.

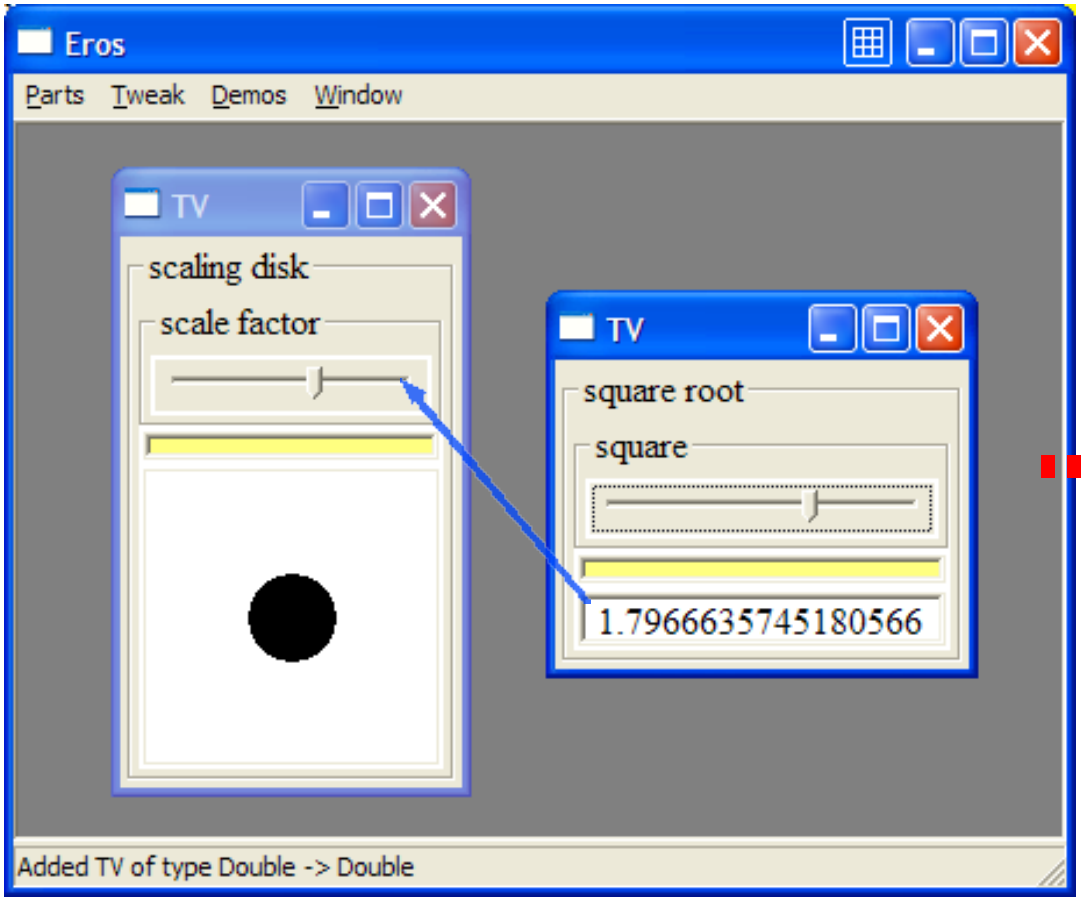
TV fusion subsumes function *application*.

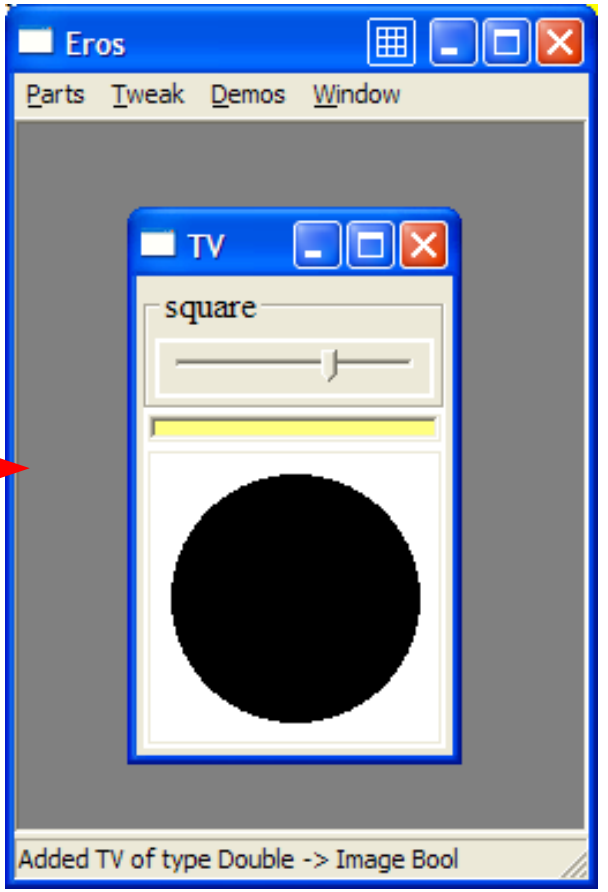




$R \to Region$ R $Region$

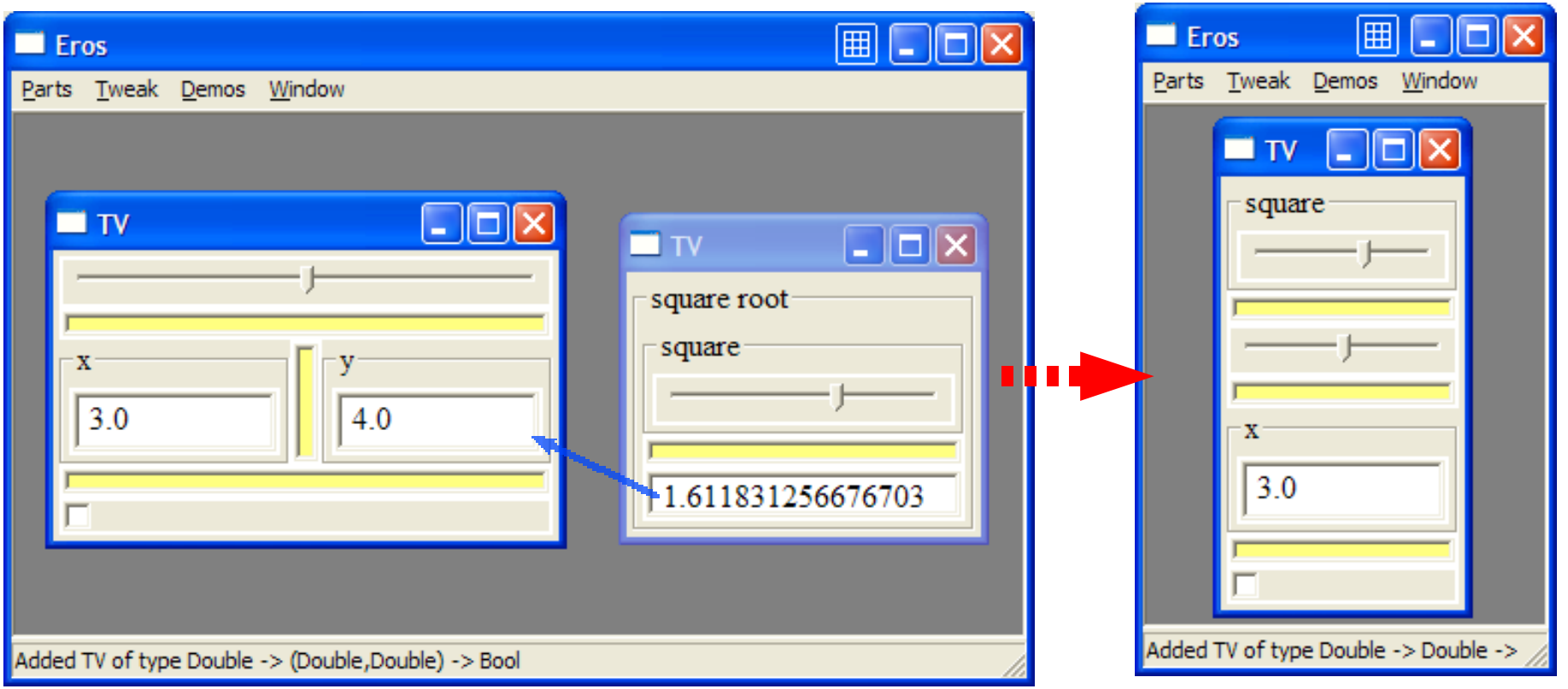
TV fusion subsumes function *composition*.





$R \to Region$ $R \to R$ $R \to Region$

Fusion may reach into nested inputs.



$R \to (R,R) \to Bool$ $\quad R \to R$ $\quad R \to R \to R \to Bool$

Let's take a look.

demo

Where are we?

- Usability and composability
- Eros user experience
- **Mechanics**

Key idea #3:

Keep visualization & value combined and separable.

```
type TV a = (Out a, a)
```

- Operate on both parts in tandem
- Combined for convenience
- Separable for composability

Visualizations *assemble*

as types and values do.

```
type Out a
put      :: Put a -> Out a
opair    :: Out a -> Out b -> Out (a, b)
olambda  :: In  a -> Out b -> Out (a->b)

type In a
get    :: Get a -> In a
ipair  :: In  a -> In b -> In (a,b)
```

Tangible values are composable MVC.

```
type Model a = a
type View  a
type Ctrl  a

type MVC a = (View a, Model a)

put      :: Put  a -> View a
opair    :: View a -> View b -> View (a, b)
olambda  :: Ctrl a -> View b -> View (a->b)

get      :: Get  a -> Ctrl a
ipair    :: Ctrl a -> Ctrl b -> Ctrl (a, b)
```

Key idea #4:

Translate gestural fusion to combinator sequences.

- "Deep application". Reaches buried
  - *arguments,*
  - *functions,* and
  - *inputs.*
- Define for values & extend to TVs.

We already have the tools to aim functions at buried arguments.

```
first  :: (a -> a') -> ((a, b) -> (a',b ))
second :: (b -> b') -> ((a, b) -> (a ,b'))
result :: (b -> b') -> ((a->b) -> (a->b'))

first        f      = \ (a, b) -> (f a,b  )
second       g      = \ (a, b) -> (a  ,g b)
result       g      = \   f    ->   g . f
```

Compositions describe type paths to edit *deeply* buried arguments.

```
sf     :: (b->b') -> (a,(b ,c))
                  -> (a,(b',c))
sf     =  second.first

frsrf :: (c->c') -> (a->(f,b->(c ,g)),e)
                  -> (a->(f,b->(c',g)),e)
frsrf =  first.result.second.result.first
```

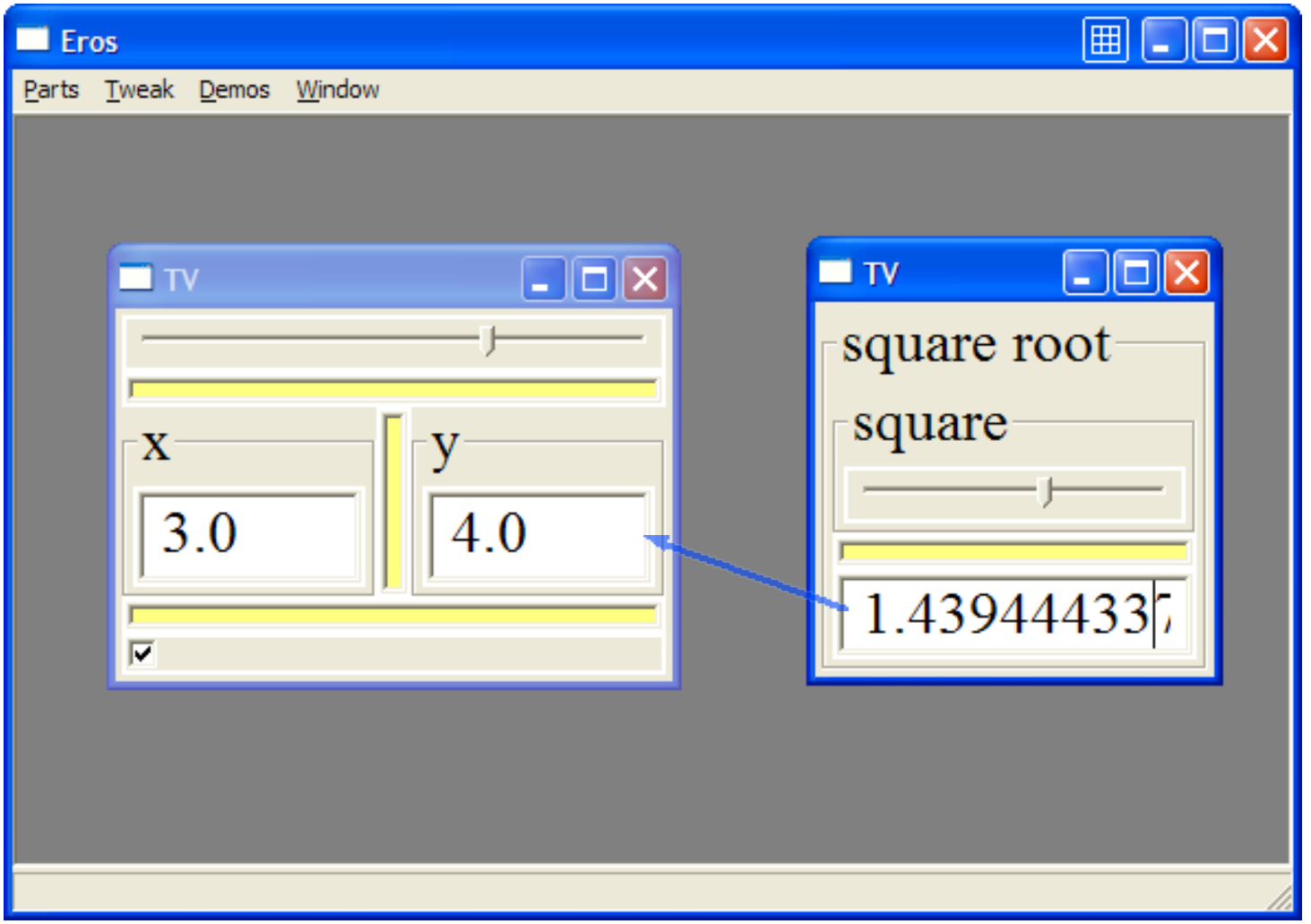
A similar game

reaches buried *functions*.

```
funFirst ::
  (d -> (c->a)) -> ((d,b) -> (c->(a,b)))
```

- Promotes a *function extractor*
- Similarly, `funSecond`, `funResult`
- Form type paths, as before.

The final combinators reach buried *inputs*.



These tools generalize.

- `first` and `second` work on arrows.
- Add **`DeepArrow`** subclass & instances for
  - visualizations & pairings,
  - types, code, etc.

We can have usability and composability.

- GUIs visualize typed values (meanings).
- Users make new TVs by fusion.
- No “universal interface” necessary.
- GUI & content combined and separable.
- Transform in tandem via deep application.

Next steps

- Run-time, optimized code generation
- GPU compiler back-end
- Map ideas to non-functional languages
- “GUIs are types” as GUI design guide
- Tangible polymorphism

Extra slides

Doug McIlroy – fast-forward to 2007

"For lovers of things small and beautiful,
http://www.cs.dartmouth.edu/~doug/powser.html
boils down basic operations on power series with numeric coefficients to the bare minimum--each is a one-liner. Included are overloaded arithmetic operators, integration, differentiation, functional composition, functional inverse and coercion from scalars. --A telling demonstration of the power of lazy evaluation and of Haskell's attunement to math."

What Doug knew

Three equivalent techniques:

- co-routines
- pipes
- lazy evaluation