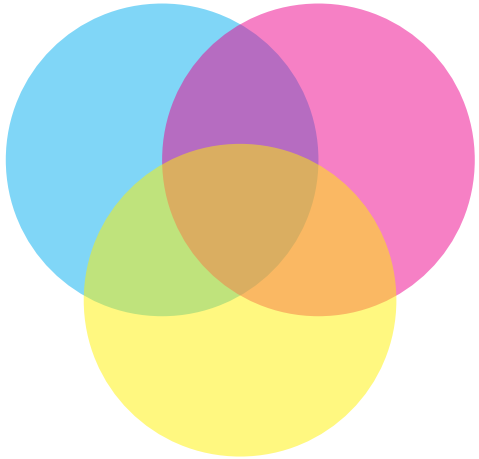


For most types, Alms infers how many times values of that type can be used (freely or only once) from the structure of the type, but in general, the domain and codomain of a function type do not determine how many times the function can be used. Thus, function arrows require annotation.

Implicit Arrow Annotations in Alms

Jesse A. Tov

Northeastern University

NEPLS
March 4, 2011

Alms:

A practical language with affine types

Affine types:

Some values can be used at most once

What's that good for?

Programming with stateful resources

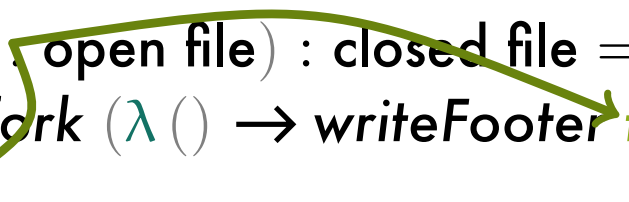
(think: generalized *typestate*)

```
let finish (f : open file) : closed file =
    let f' = writeFooter f
     in close f'
```

```
let finish (f : open file) : closed file =
    let f' = writeFooter f
     in close f'

let finish (f : open file) : closed file =
    Thread.fork (λ () → writeFooter f);
    close f
```

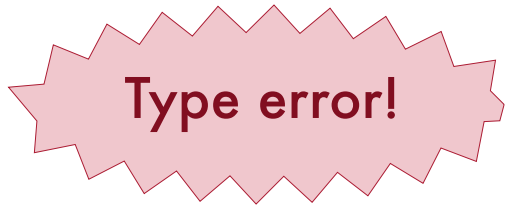
```
let finish (f : open file) : closed file =
    let f′ = writeFooter f
     in close f′
```

```
let finish (f : open file) : closed file =
    Thread.fork (λ () → writeFooter f);
    close f
```



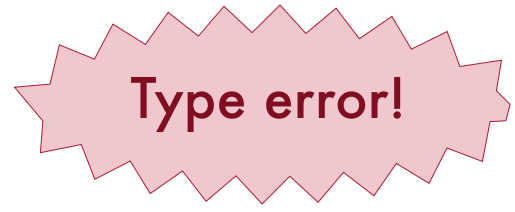
Type error!

```
let finish (f : open file) : closed file =
    let f′ = writeFooter f
     in close f′

let finish (f : open file) : closed file =
    Thread.fork (λ () → writeFooter f);
    close f

let square (z : int) = z × z
```



```
let finish (f : open file) : closed file =
    let f' = writeFooter f
     in close f'
```

```
let finish (f : open file) : closed file =
    Thread.fork (λ () → writeFooter f);
    close f
```




```
let square (z : int) = z × z
```

Kinds:

int : U

α file : A

Kinds:

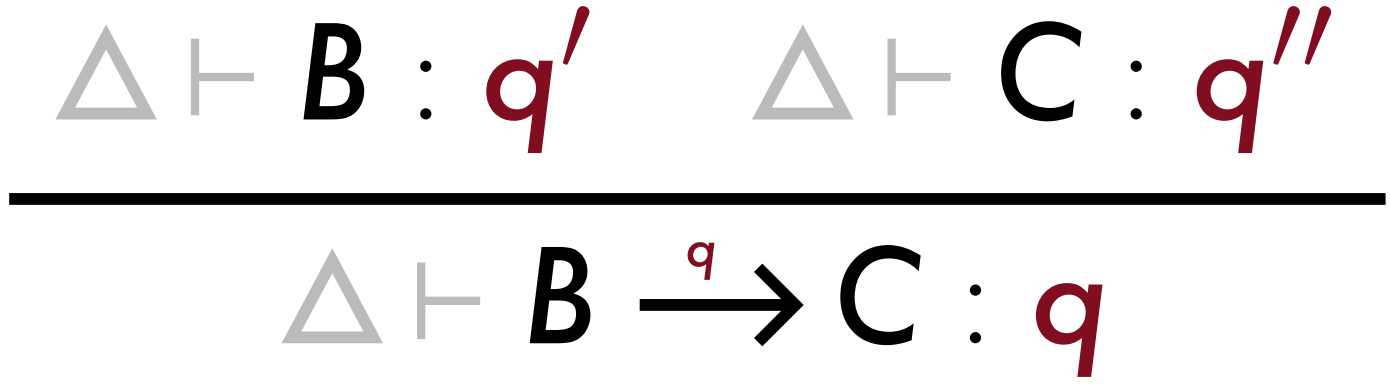
$$q \in \begin{array}{c} A \\ | \\ U \end{array}$$

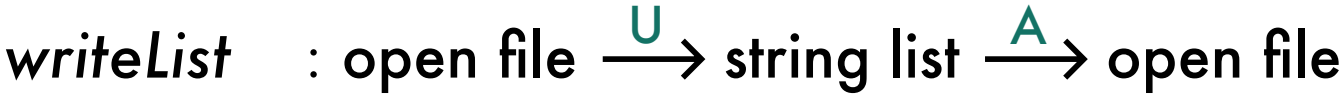
$$\frac{\Delta \vdash B : q}{\Delta \vdash B \text{ list} : q} \qquad \frac{\Delta \vdash B : q \quad \Delta \vdash C : q'}{\Delta \vdash B \times C : q \sqcup q'}$$

$$\frac{\Delta \vdash B : q' \qquad \Delta \vdash C : q''}{\Delta \vdash B \to C : ?}$$

```
let later (name : string) : unit ⟶ open file =
    λ () → open name

let now (name : string) : unit ⟶ open file =
    let f = open name
     in λ () → f
```

```
let later (name : string) : unit -U-> open file =
  λ() → open name

let now (name : string) : unit -A-> open file =
  let f = open name
   in λ() → f
```


Δ ⊢ B : q′
Δ ⊢ C : q″
Δ ⊢ B →q C : q


writeList : open file U string list A open file

writeList : open file $\xrightarrow{U}$ string list $\xrightarrow{A}$ open file

writeList f vs. *f*

writeList : open file $\xrightarrow{\text{U}}$ string list $\xrightarrow{\text{A}}$ open file

writeList f vs. *f*

compose : $(\beta \xrightarrow{\delta} \gamma) \xrightarrow{\text{U}} (\alpha \xrightarrow{\epsilon} \beta) \xrightarrow{\delta} \alpha \xrightarrow{\delta \sqcup \epsilon} \gamma$

compose f vs. *f*

compose f g vs. *f* and *g*

writeList : open file $\xrightarrow{U}$ string list $\xrightarrow{A}$ open file

writeList f vs. *f*

compose : $(\beta \xrightarrow{\delta} \gamma) \xrightarrow{U} (\alpha \xrightarrow{\epsilon} \beta) \xrightarrow{\delta} \alpha \xrightarrow{\delta \sqcup \epsilon} \gamma$

compose *f* vs. *f*

compose *f g* vs. *f* and *g*

writeList : open file $\xrightarrow{U}$ string list $\xrightarrow{A}$ open file

writeList f vs. *f*

compose : $(\beta \xrightarrow{\delta} \gamma) \xrightarrow{U} (\alpha \xrightarrow{\epsilon} \beta) \xrightarrow{\delta} \alpha \xrightarrow{\delta \sqcup \epsilon} \gamma$

compose *f* vs. *f*

compose *f g* vs. *f* and *g*

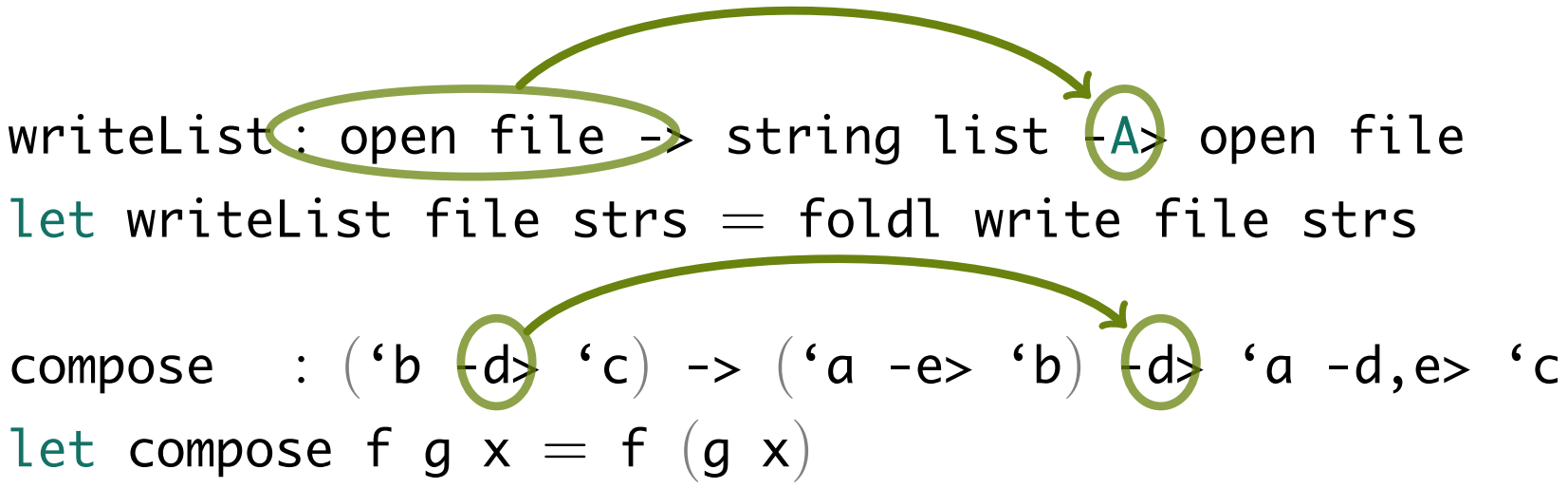
```
writeList : open file -U> string list -A> open file

compose   : (‘b -d> ‘c) -U> (‘a -e> ‘b) -d> ‘a -d,e> ‘c
```

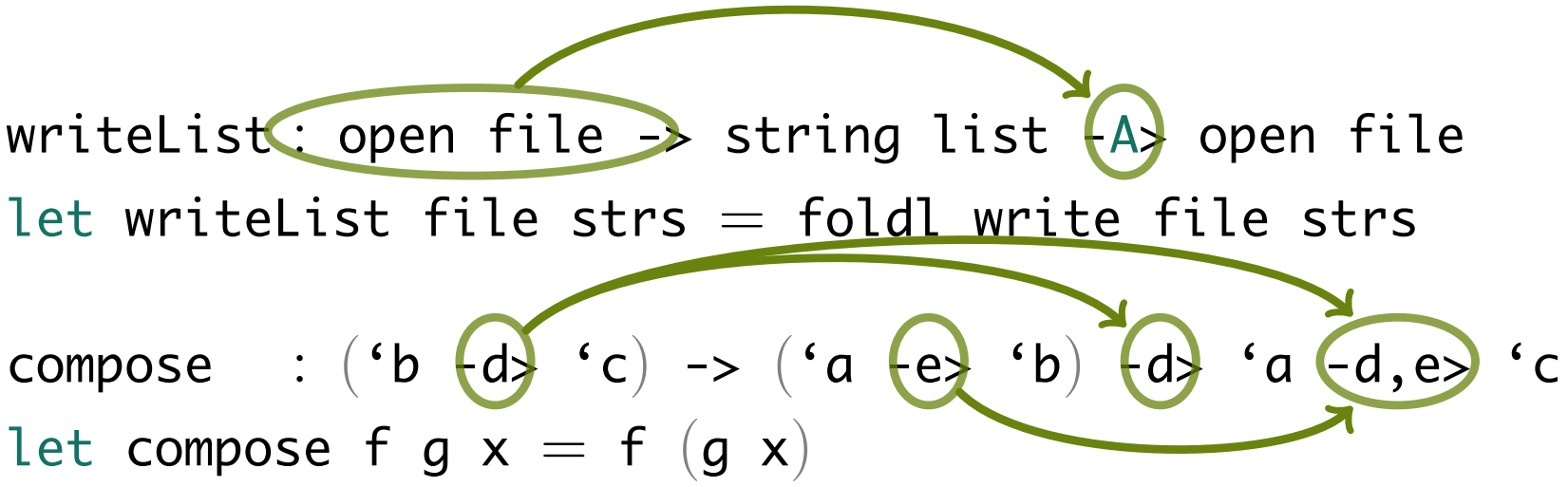
```
writeList : open file -> string list -A> open file

compose   : (‘b -d> ‘c) -> (‘a -e> ‘b) -d> ‘a -d,e> ‘c
```

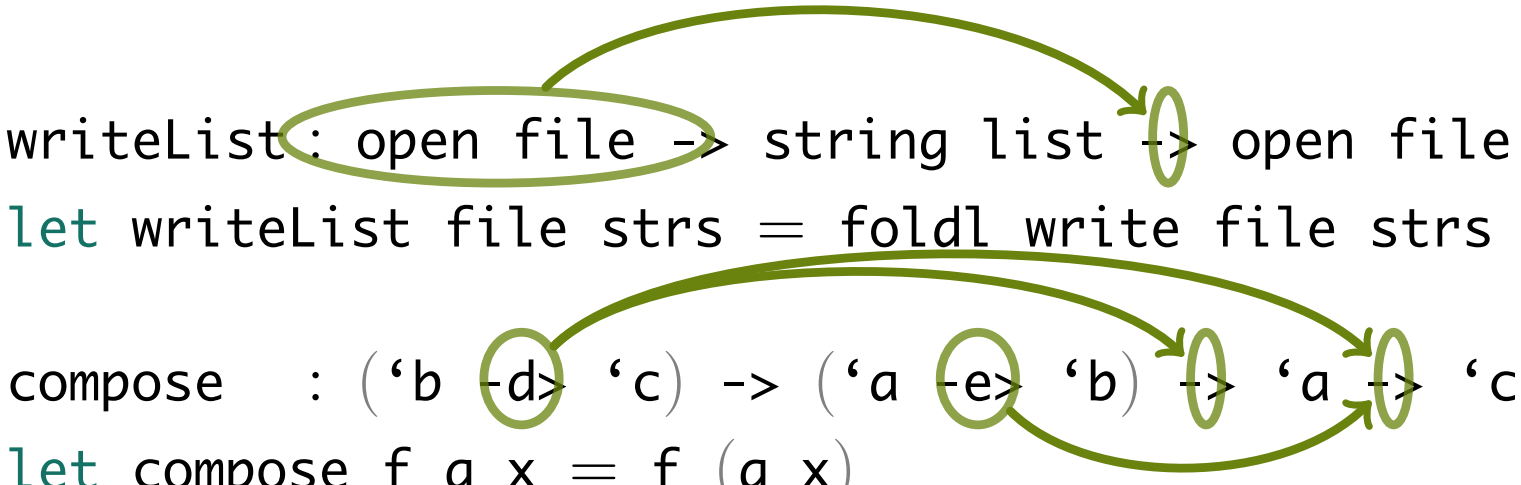
```
writeList : open file -> string list -A> open file

compose   : (‘b -d> ‘c) -> (‘a -e> ‘b) -d> ‘a -d,e> ‘c
```

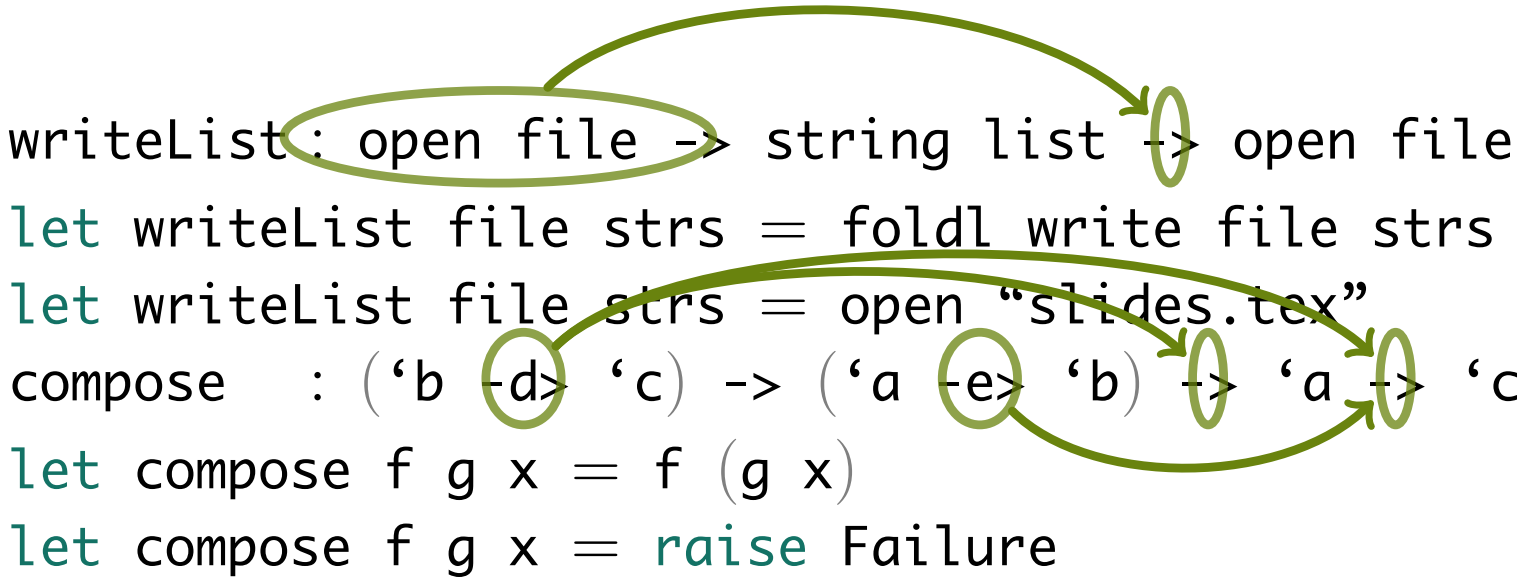
```
writeList : open file -> string list -A> open file
let writeList file strs = foldl write file strs

compose   : (‘b -d> ‘c) -> (‘a -e> ‘b) -d> ‘a -d,e> ‘c
let compose f g x = f (g x)
```

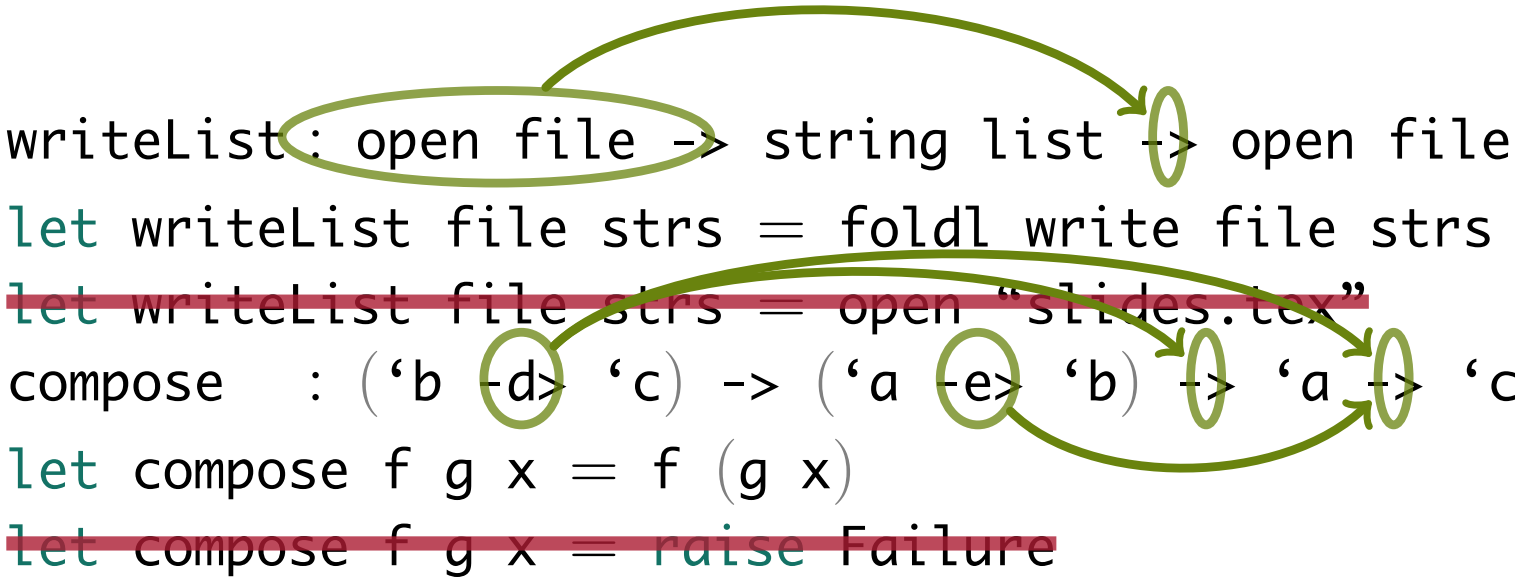
```
writeList : open file -> string list -A> open file
let writeList file strs = foldl write file strs

compose   : ('b -d> 'c) -> ('a -e> 'b) -d> 'a -d,e> 'c
let compose f g x = f (g x)
```


writeList : open file -> string list -A> open file
let writeList file strs = foldl write file strs
compose : ('b -d> 'c) -> ('a -e> 'b) -d> 'a -d,e> 'c
let compose f g x = f (g x)


writeList : open file -> string list -A> open file
let writeList file strs = foldl write file strs
compose : ('b -d> 'c) -> ('a -e> 'b) -d> 'a -d,e> 'c
let compose f g x = f (g x)


writeList : open file -> string list -> open file
let writeList file strs = foldl write file strs
compose : ('b -d> 'c) -> ('a -e> 'b) -> 'a -> 'c
let compose f g x = f (g x)


writeList : open file -> string list -> open file
let writeList file strs = foldl write file strs
let writeList file strs = open "slides.tex"
compose : ('b -d> 'c) -> ('a -e> 'b) -> 'a -> 'c
let compose f g x = f (g x)
let compose f g x = raise Failure


writeList : open file -> string list -> open file
let writeList file strs = foldl write file strs
let writeList file strs = open "slides.tex"
compose : ('b -d> 'c) -> ('a -e> 'b) -> 'a -> 'c
let compose f g x = f (g x)
let compose f g x = raise Failure

Alms's Standard Library:

Rule	№ of Annotations
explicit	588
implicit Us	63
new rule	20

Alms's Standard Library:

Rule	№ of Annotations
explicit	588
implicit Us	63
new rule	20

17 have negative As
1 relates domain and range
2 for a weird contract thing

Try Alms:

http://www.ccs.neu.edu/~tov/pubs/alms

(or Google: `alms affine`)