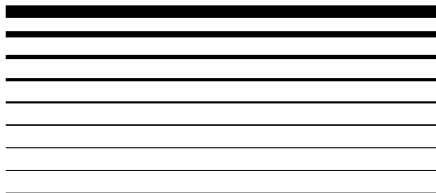


For most types, Alms infers how many times values of that type can be used (freely or only once) from the structure of the type, but in general, the domain and codomain of a function type do not determine how many times the function can be used. Thus, function arrows require annotation.



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*)

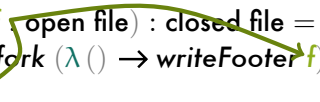
```
let finish (f : open file) : closed file =  
  let f' = writeFooter f  
  in close f'
```

```
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'
```

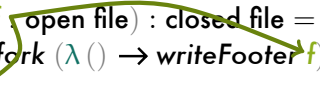
```
let finish (f : open file) : closed file =  
  Thread.fork (λ () → writeFooter f);  
  close f
```



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
```



Type error!

```
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:

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

$$\frac{\Delta \vdash B : q}{\Delta \vdash B \text{ list} : q}$$

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

$$\Delta \vdash B : q' \quad \Delta \vdash C : q''$$

$$\Delta \vdash B \rightarrow C : ?$$

let *later* (*name* : string) : unit \longrightarrow open file =
 $\lambda () \rightarrow$ *open name*

let *now* (*name* : string) : unit \longrightarrow open file =
 let *f* = *open name*
 in $\lambda () \rightarrow$ *f*

let *later* (*name* : string) : unit \xrightarrow{U} open file =
λ () → open name

let *now* (*name* : string) : unit \xrightarrow{A} open file =
let *f* = open name
in λ () → *f*

$$\Delta \vdash B : q' \quad \Delta \vdash C : q''$$

$$\Delta \vdash B \xrightarrow{q} C : q$$

writeList : open file \xrightarrow{U} string list \xrightarrow{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)

`writelnList: open file -> string list -A> open file`
`let writelnList 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)
```

The diagram illustrates lambda abstraction in Haskell code. In the first line, the lambda argument 'open file' is circled in green, and a green arrow points from it to the 'open file' argument of the 'foldl' function. In the second line, the lambda arguments 'b', 'c', 'a', 'e', and 'b' are circled in green. Green arrows show 'b' binding to 'a', 'c' binding to 'b', 'a' binding to 'a', and 'e' binding to 'b' in the function application 'f (g x)'. The lambda argument 'a' is also circled in green, and a green arrow points from it to the 'a' argument of the 'compose' function.

```
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
```

The diagram illustrates function composition and lambda expressions in Haskell code. It features several green circles and arrows highlighting specific parts of the code:

- A circle around the lambda argument `open file` in the first line, with an arrow pointing to the lambda return value `open file`.
- A circle around the lambda argument `'b` in the second line, with an arrow pointing to the lambda return value `'a`.
- A circle around the lambda argument `'c` in the second line, with an arrow pointing to the lambda return value `'b`.
- A circle around the lambda argument `'a` in the second line, with an arrow pointing to the lambda return value `'c`.


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

Alms's Standard Library:

Rule	No 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)