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
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'

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 \( \text{finish} \ (f : \text{open file}) : \text{closed file} = \)
let \( f' = \text{writeFooter} \ f \)
in close \( f' \)

let \( \text{finish} \ (f : \text{open file}) : \text{closed file} = \)
Thread.fork \((\lambda () \rightarrow \text{writeFooter} \ f)\); 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'

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:

$$q \in U$$

$$\Delta \vdash B : q$$

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

$$\Delta \vdash B \times C : q \sqcup q'$$
\[ \Delta \vdash B : q' \quad \Delta \vdash C : q'' \]

\[ \Delta \vdash B \rightarrow C : ? \]
let later (name : string) : unit → 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 \rightarrow open file =
\lambda () \rightarrow open name

let now (name : string) : unit A \rightarrow open file =
let f = open name
let \lambda () \rightarrow f
\[ \text{writeList} : \text{open file} \xrightarrow{U} \text{string list} \xrightarrow{A} \text{open file} \]
\texttt{writeList} : open file $\xrightarrow{\text{U}}$ string list $\xrightarrow{\text{A}}$ open file

\texttt{writeList f} \quad \text{vs.} \quad f
\textbf{writeList} : open file $\xrightarrow{U} \text{string list} \xrightarrow{A} \text{open file}$

\textbf{writeList $f$} vs. $f$

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

\textbf{compose $f$} vs. $f$

\textbf{compose $f \circ g$} vs. $f$ and $g$
**writeList**: open file \(\xrightarrow{\cup}\) string list \(\xrightarrow{\mathcal{A}}\) open file

\[
\text{writeList } f \quad \text{vs.} \quad f
\]

**compose**: \((\beta \xrightarrow{\delta} \gamma) \xrightarrow{\cup} (\alpha \xrightarrow{\epsilon} \beta) \xrightarrow{\delta} \alpha \xrightarrow{\delta \sqcup \epsilon} \gamma\)

\[
\text{compose } f \quad \text{vs.} \quad f
\]

\[
\text{compose } f \ g \quad \text{vs.} \quad f \text{ and } g
\]
**writeList** : open file $\xrightarrow{U}$ string list $\xrightarrow{A}$ open file

writeList $f$ vs. $f$

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

compose $f$ vs. $f$

compose $f \circ g$ 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 -> open file

compose : (‘b -d> ‘c) -> (‘a -e> ‘b) -d> ‘a -d,e> ‘c
writeList : open file -> string list -A> open file

compose : ('b → 'c) -> ('a → 'b) → 'a → '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 -> open file
let writeList file strs = foldl write file strs

compose : ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c
let compose f g x = f (g x)
writeList : open file \rightarrow string list \rightarrow open file

let writeList file strs = foldl write file strs

compose : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow b, e \rightarrow 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 -> 'c) -> ('a -> '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 → ‘c) -> (‘a → ‘b) -> ‘a -> ‘c
let compose f g x = f (g x)
let compose f g x = raise Failure
writeList : open file -> string list list -> open file
let writeList file strs = foldl write file strs

let compose : ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c
let compose f g x = f (g x)

let compose f g x = raise Failure
<table>
<thead>
<tr>
<th>Rule</th>
<th>№ of Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit</td>
<td>588</td>
</tr>
<tr>
<td>implicit Us</td>
<td>63</td>
</tr>
<tr>
<td>new rule</td>
<td>20</td>
</tr>
</tbody>
</table>
Alms’s Standard Library:

<table>
<thead>
<tr>
<th>Rule</th>
<th>No of Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit</td>
<td>588</td>
</tr>
<tr>
<td>implicit Us</td>
<td>63</td>
</tr>
<tr>
<td>new rule</td>
<td>20</td>
</tr>
</tbody>
</table>

17 have negative A
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)