

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Function Subtyping

Now functions

- Already know a caller can use subtyping for arguments passed
 - Or on the result
- More interesting: When is one function type a subtype of another?
 - Important for higher-order functions: If a function expects an argument of type $t_1 \rightarrow t_2$, can you pass a $t_3 \rightarrow t_4$ instead?
 - Coming next: Important for understanding methods
 - (An object type is a lot like a record type where “method positions” are immutable and have function types)

Example

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
                p : {x:real,y:real}) =
  let val p2 : {x:real,y:real} = f p
      val dx : real = p2.x - p.x
      val dy : real = p2.y - p.y
  in Math.sqrt(dx*dx + dy*dy) end

fun flip p = {x = ~p.x, y=~p.y}
val d = distMoved(flip, {x=3.0, y=4.0})
```

No subtyping here yet:

- `flip` has exactly the type `distMoved` expects for `f`
- Can pass in a record with extra fields for `p`, but that's old news

Return-type subtyping

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
  let val p2 : {x:real,y:real} = f p
      val dx : real = p2.x - p.x
      val dy : real = p2.y - p.y
  in Math.sqrt(dx*dx + dy*dy) end

fun flipGreen p = {x = ~p.x, y=~p.y, color="green"}
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```

- Return type of `flipGreen` is `{x:real,y:real,color:string}`, but `distMoved` expects a return type of `{x:real,y:real}`
- Nothing goes wrong: **If** $ta <: tb$, **then** $t \rightarrow ta <: t \rightarrow tb$
 - A function can return “*more than it needs to*”
 - Jargon: “Return types are *covariant*”

This is wrong

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
  let val p2 : {x:real,y:real} = f p
      val dx : real = p2.x - p.x
      val dy : real = p2.y - p.y
  in Math.sqrt(dx*dx + dy*dy) end

fun flipIfGreen p = if p.color = "green" (*kaboom!*)
                    then {x = ~p.x, y=~p.y}
                    else {x = p.x, y=p.y}

val d = distMoved(flipIfGreen, {x=3.0, y=4.0})
```

- Argument type of `flipIfGreen` is `{x:real,y:real,color:string}`, but it is called with a `{x:real,y:real}`
- Unsound! `ta <: tb` does **NOT** allow `ta -> t <: tb -> t`

The other way works!

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
  let val p2 : {x:real,y:real} = f p
      val dx : real = p2.x - p.x
      val dy : real = p2.y - p.y
  in Math.sqrt(dx*dx + dy*dy) end

fun flipX_Y0 p = {x = ~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

- Argument type of `flipX_Y0` is `{x:real}` but it is called with a `{x:real,y:real}`, which is fine
- If $tb <: ta$, then $ta \rightarrow t <: tb \rightarrow t$
 - A function can assume “less than it needs to” about arguments
 - Jargon: “Argument types are *contravariant*”

Can do both

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
  let val p2 : {x:real,y:real} = f p
      val dx : real = p2.x - p.x
      val dy : real = p2.y - p.y
  in Math.sqrt(dx*dx + dy*dy) end

fun flipXMakeGreen p = {x = ~p.x, y=0.0, color="green"}
val d = distMoved(flipXMakeGreen, {x=3.0, y=4.0})
```

- `flipXMakeGreen` has type
 $\{x:real\} \rightarrow \{x:real,y:real,color:string\}$
- Fine to pass a function of such a type as function of type
 $\{x:real,y:real\} \rightarrow \{x:real,y:real\}$
- If $t3 <: t1$ and $t2 <: t4$, then $t1 \rightarrow t2 <: t3 \rightarrow t4$

Conclusion

- If $t3 <: t1$ and $t2 <: t4$, then $t1 \rightarrow t2 <: t3 \rightarrow t4$
 - Function subtyping contravariant in argument(s) and covariant in results
- Also essential for understanding subtyping and methods in OOP
- The most unintuitive concept in this course
 - Smart people often forget and convince themselves that covariant arguments are okay
 - These smart people are always mistaken
 - At times, you or your boss or your friend may do this
 - Remember: A guy with a PhD in PL ***jumped out and down*** insisting that function/method subtyping is always contravariant in its argument -- covariant is unsound