

A Simplified Calculus of Higher-Order Modules

Paul Govereau
govereau@eecs.harvard.edu

This Talk

Large scale structure

Separate compilation

Type **abstraction** and **generativity**

Phase separation

First vs. second class modules

Type sharing

...

Type Abstraction

`Sym :: SYM` “sealing `Sym` with `SYM`”

Principal signature does not have abstract types

`SYM` may hide definitions of types in `Sym`
“**abstract**”

Different instances of `Sym` may not be able to share hidden types “**generative**”

Abstraction and Generativity

M,N

```
type t = int
```

S

```
type t
```

M :: S N :: S

abstract

M :: S

```
type t = S.t
```

N :: S

```
type t = S.t
```

generative

M :: S

```
type t = M.t
```

N :: S

```
type t = N.t
```

Generativity in Signatures

- Represent **abstract** and **generative** types in signatures
- Abstract types : **type** t
- Generative types : **newtype** t
- Signatures may contain **type** and **newtype**

type and newtype

	Abstract?	Generative?
<code>type t = int</code>	No	No
<code>type t</code>	Yes	No
<code>newtype t</code>	Yes	Yes

SML-style generative functor

```
funsig FS (A : ...) = sig  
  newtype t  
  newtype u  
  type v = int  
  ...  
end
```

```
structure F :: FS  
  F(A).t ≠ F(A).t
```

OCaml-style applicative functor

```
funsig FS (A : ...) = sig  
  type t  
  type u  
  type v = int  
  ...  
end
```

```
structure F :: FS  
  F(A).t = F(A).t
```

Symbol Table Module

```
signature SYM = sig
  type sym      = int * string
  type extsym  = string
  val new       : string -> sym
  val toString : sym -> string
  val export    : sym -> extsym
  val import    : extsym -> sym
end
```

Symbol Table Module

```
structure S1 = Sym :: SYM
```

```
let x1 = S1.new()  
    ex1 = S1.export x1  
in print ex1
```

Error!

```
let x1 = S1.new()  
    ex1 = S1.export x1  
in print (S1.toString ex1)
```

Ok.

Symbol Table Module

```
structure S1 = Sym :: SYM  
structure S2 = Sym :: SYM
```

```
let x1 = S1.new()  
in S2.toString x1
```

Error!

```
let x1 = S1.new()  
in S2.import(S1.export x1)
```

Ok.

The Type of Sym

```
signature SYM = sig
  type sym      = int * string
  type extsym   = string
  val new       : string -> sym
  val toString  : sym -> string
  val export    : sym -> extsym
  val import    : extsym -> sym
end
```

The Type of Sym

```
signature SYM = sig
  type sym      = int * string
  type extsym
  val new       : string -> sym
  val toString  : sym -> string
  val export    : sym -> extsym
  val import    : extsym -> sym
end
```

`extsym` is an **abstract** and applicative type

The Type of Sym

```
signature SYM = sig
  newtype sym
  type extsym
  val new      : string -> sym
  val toString: sym -> string
  val export   : sym -> extsym
  val import   : extsym -> sym
end
```

sym is **abstract** and **generative**

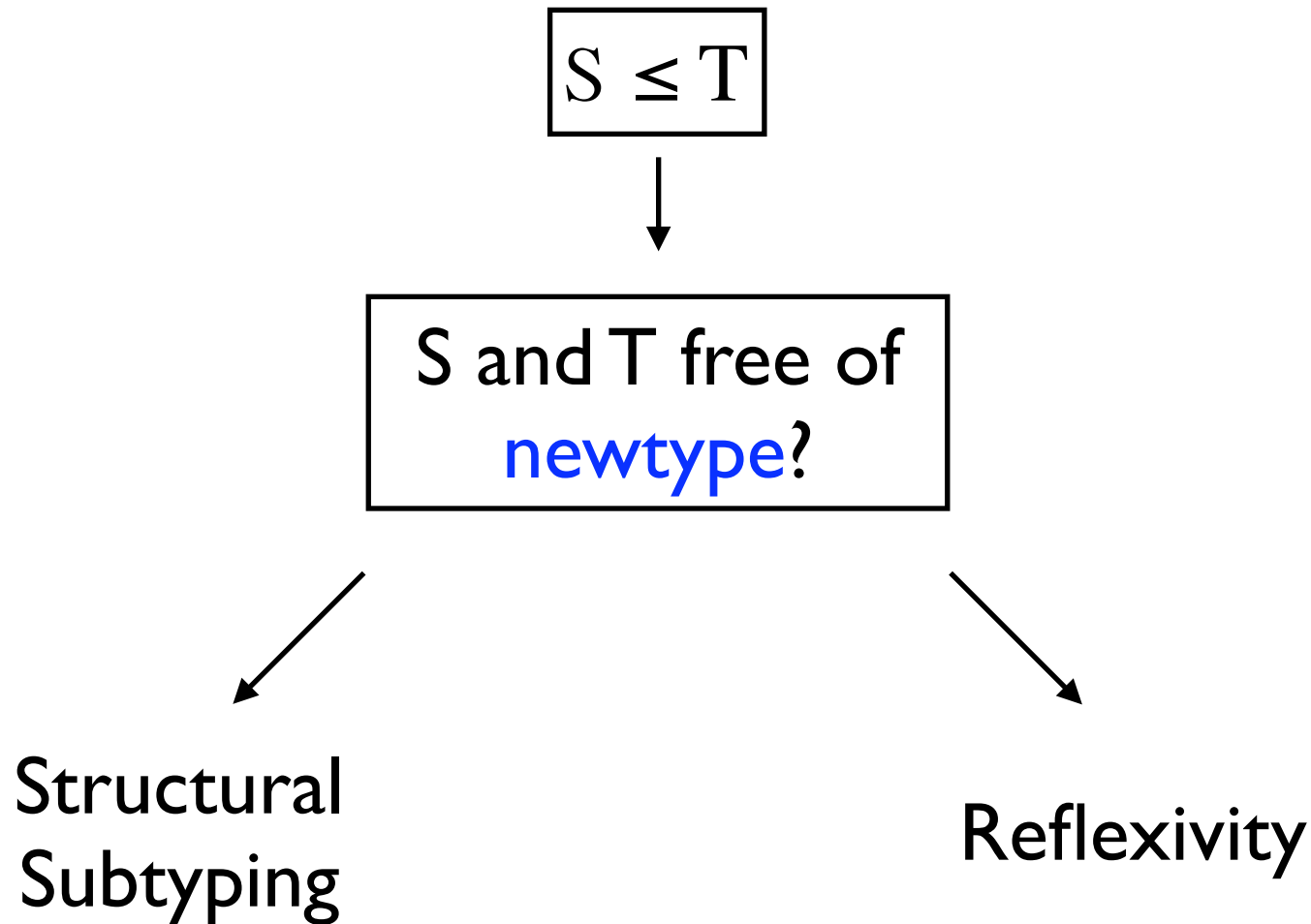
Subtyping

$\tau \leq \text{type } t$

$\text{type } t \leq \text{newtype } t$

Principal signatures do not have abstract or generative types

Simplified View of \leq



Beyond ML Modules

```
signature SYM = sig
  newtype sym
  val new      : string -> sym
  val toString: sym -> string
end
```

```
structure S1 :: SYM = struct
  type sym      = int * string
  fun new str   = ...
  val toString  = snd
end
```

```
structure S2 :: SYM = struct
  type sym      = string * string
  fun new str   = ...
  val toString  = snd
end
```

Beyond ML Modules

```
structure Common = struct
  type sym      =  $\exists a.(a*\text{string})$ 
  val toString = snd
end
```

```
structure S1 :: SYM = struct
  type sym      = int * string
  ...
end
```

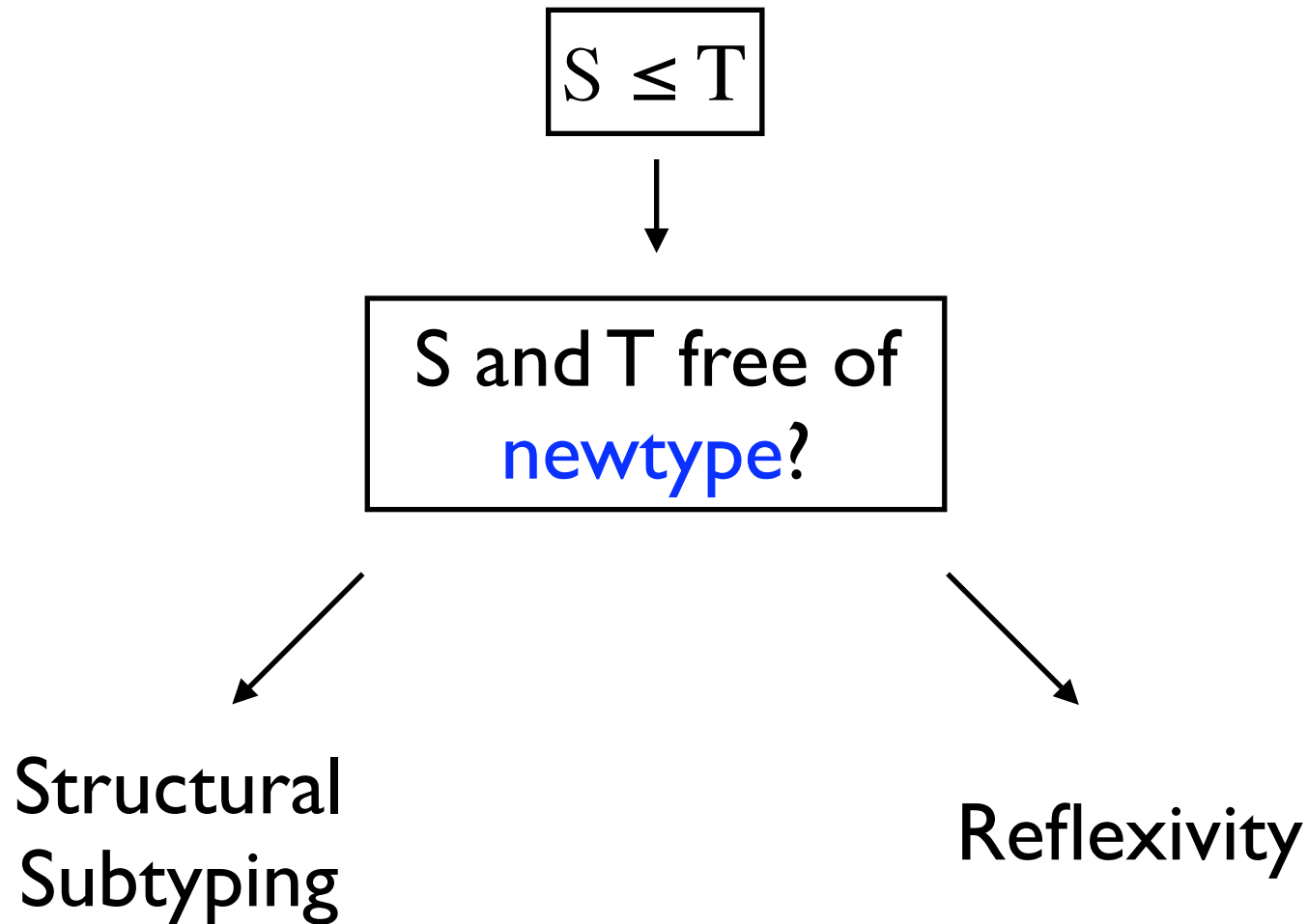
This does not work!

Beyond ML Modules

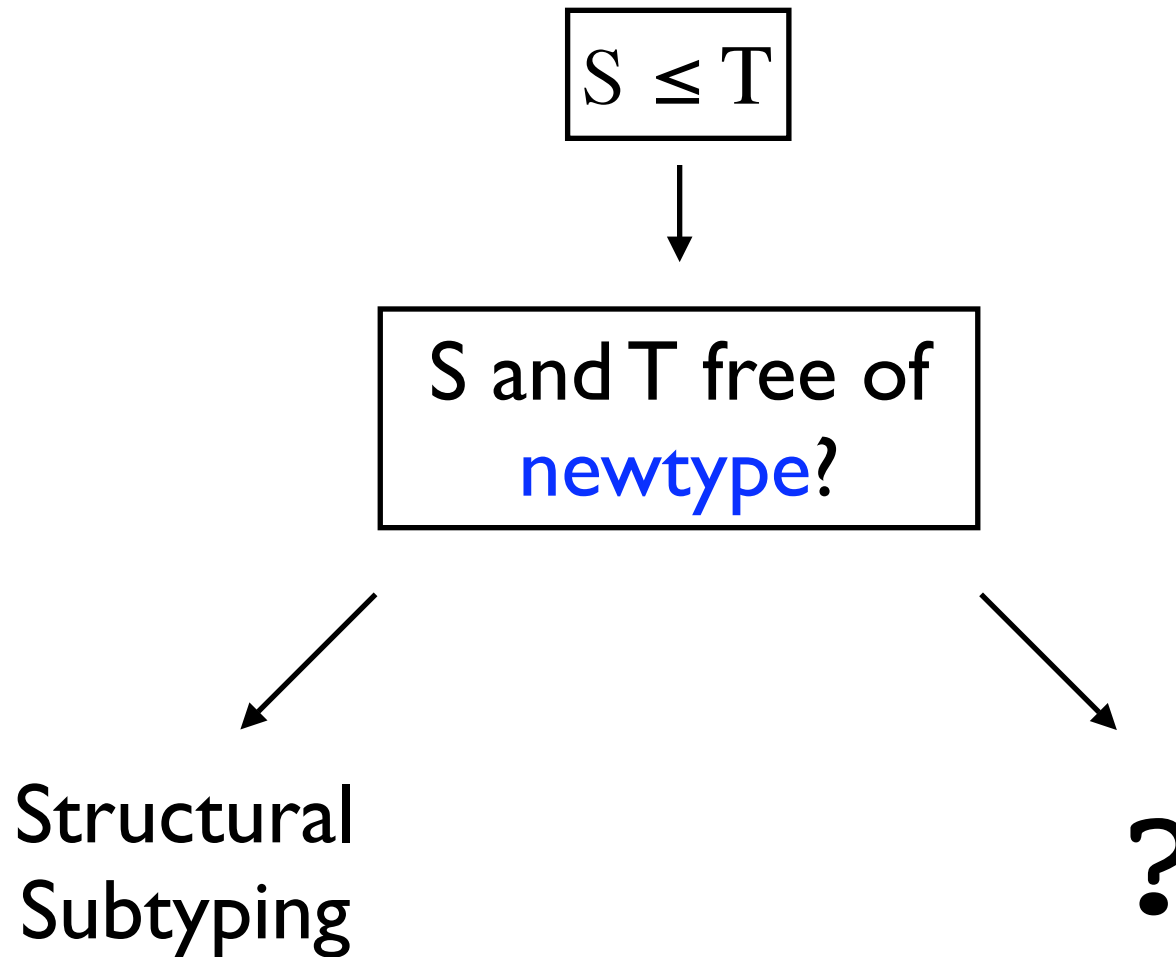
```
structure Common = struct
  type sym      =  $\exists a.(a*\text{string})$ 
  val toString = snd
end
```

```
structure S1 :: SYM = struct
  type sym  $\leq$  Common.sym = int * string
  ...
end
```

Simplified View of \leq



Simplified View of \leq



Why Simplified?

(Compared to Dryer, Crary and Harper '03)

- Dependent products/sums and singletons
- Two flavors of types within signatures
- Extensible subtype relation

- One kind of functor/sealing
- No effect system

Thank You.

Paul Govereau
govereau@eecs.harvard.edu