Phantom Types and Subtyping

Riccardo Pucella
Northeastern University

(Joint work with Matthew Fluet, TTI Chicago)

Sun Microsystems April 5, 2007

Phantom Types

Essentially a typing trick to get a Hindley-Milner type system to do something it was not designed to do

- Early uses in Standard ML and Haskell to give a statically safe interface to some unsafe APIs

Relies on:

- parametric polymorphism
- type equivalence properties

An example is the easiest way to explain...

Example: Typed Expressions

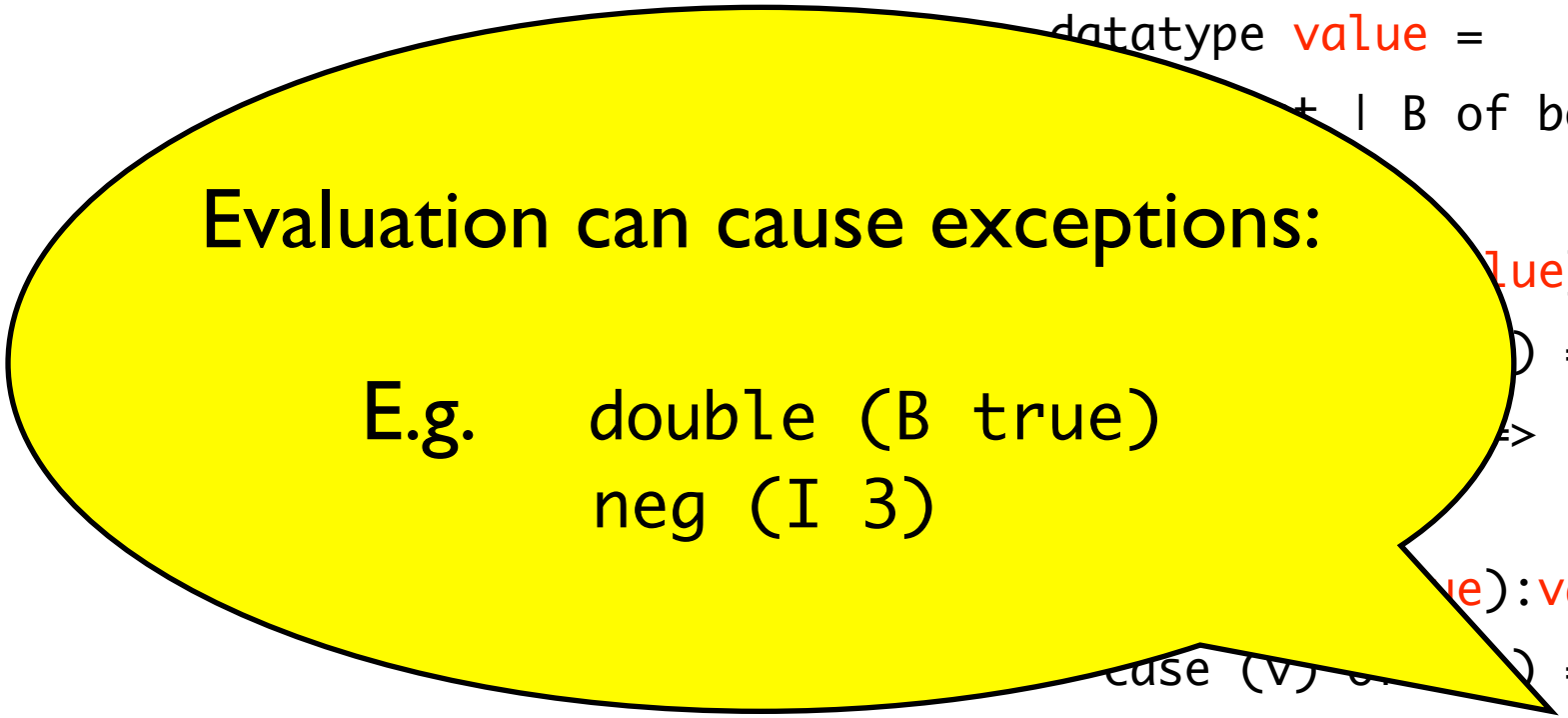
```
datatype value =
  I of int | B of bool

fun double (v:value):value =
  case (v) of I(i) => I(i*2)
            | _ => raise TypeError

fun neg (v:value):value =
  case (v) of B(b) => B(not b)
            | _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Example: Typed Expressions




```
datatype value =
         | B of bool

lue):value =
) => I(i*2)
=> raise TypeError

ue):value =
) => B(not b)
| _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Example: Typed Expressions

Can we ensure statically that no exception is raised?

Sure...

SOLUTION I

Introduce different types for values

```
datatype value =
  I of int | B of bool

fun double (v:value):value =
  case (v) of I(i) => I(i*2)
            | _ => raise TypeError

fun neg (v:value):value =
  case (v) of B(b) => B(not b)
            | _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Example: Typed Expressions

```
datatype ivalue = I' of value
datatype bvalue = B' of value

fun mkI (i:int):ivalue = I'(I(i))
fun mkB (b:bool):bvalue = B'(B(b))

fun sdouble (v:ivalue):ivalue =
  I'(double (I'of (v)))
fun sneg (v:bvalue):bvalue =
  B'(neg (B'of (v)))

fun stoStringI (v:ivalue):string =
  toString (I'of (v))
fun stoStringB (v:bvalue):string =
  toString (B'of (v))
```

```
datatype value =
  I of int | B of bool

fun double (v:value):value =
  case (v) of I(i) => I(i*2)
            | _ => raise TypeError

fun neg (v:value):value =
  case (v) of B(b) => B(not b)
            | _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Example: Typed Expressions

```
datatype ivalue = I' of value
datatype bvalue = B' of value

fun mkI (i:int):ivalue =
fun mkB (b:bool):bvalue

fun sdouble (v:ivalue):
  I'(double (I'of (v)))
fun sneg (v:bvalue):bvalue =
  B'(neg (B'of (v)))

fun stoStringI (v:ivalue):string =
  toString (I'of (v))
fun stoStringB (v:bvalue):string =
  toString (B'of (v))
```

```
datatype value =
  I of int | B of bool

                    lue =
                    2)
                    TypeError

                  ):value =
  case (v) of B(b) => B(not b)
            | _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Types wrapping around the underlying primitive type

Example: Type

"Safe" version of operations

```
sdouble (mkB true)
sneg (mkI 3)
```

do not type check

```
datatype ivalue = I' of val
datatype bvalue = B' of va

fun mkI (i:int):ivalue = I'(
fun mkB (b:bool):bvalue = B'(

fun sdouble (v:ivalue):ival
  I'(double (I'of (v)))
fun sneg (v:bvalue):bvalue =
  B'(neg (B'of (v)))

fun stoStringI (v:ivalue):string =
  toString (I'of (v))
fun stoStringB (v:bvalue):string =
  toString (B'of (v))
```

```
                                  se TypeError

fun neg (v:value):value =
  case (v) of B(b) => B(not b)
            | _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Example: Typed Expressions

```
datatype ivalue = I' of value
datatype bvalue = B' of value

fun mkI (i:int):ivalue = I'(I
fun mkB (b:bool):bvalue = B'

fun sdouble (v:ivalue):ival
  I'(double (I'of (v)))
fun sneg (v:bvalue):bvalue =
  B'(neg (B'of (v)))

fun stoStringI (v:ivalue):st
  toString (I'of (v))
fun stoStringB (v:bvalue):string =
  toString (B'of (v))
```

Requires two forms of `stoString`

Or forces you to cast an `ivalue` or `bvalue` to a value before calling `stoString`

```
datatyp
                                    Error

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Example: Typed Expressions

```
datatype ivalue = I' of value
datatype bvalue = B' of value

fun mkI (i:int):ivalue = I'(I(i))
fun mkB (b:bool):bvalue = B'(B(b))

fun sdouble (v:ivalue):ivalue =
  I'(double (I'of (v)))
fun sneg (v:bvalue):bvalue =
  B'(neg (B'of (v)))

fun stoStringI (v:ivalue):string =
  toString (I'of (v))
fun stoStringB (v:bvalue):string =
  toString (B'of (v))
```

```
datatype value =
  I of int | B of bool

fun double (v:value):value =
  case (v) of I(i) => I(i*2)
            | _ => raise TypeError

fun neg (v:value):value =
  case (v) of B(b) => B(not b)
            | _ => raise TypeError

fun toString (v:value):string =
  case (v)
   of I(i) => Int.toString(i)
    | B(b) => Bool.toString(b)
```

SOLUTION 2

Introduce different types for values

But put the "tag" in the type in such a way that we can be polymorphic in the tag

and make sure we share the underlying representation

Example: Typed Expressions

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of value

fun mkI (i:int):IT svalue = S(I(i))
fun mkB (b:bool):BT svalue = S(B(b))

fun sdouble (v:IT svalue):IT svalue =
  S(double (Sof (v)))
fun sneg (v:BT svalue):BT svalue =
  S(neg (Sof (v)))

fun stoString (v:α svalue):string =
  S(toString (Sof (v)))
```

```
datatype value =
  I of int | B of bool

fun double (v:value):value =
  case (v) of I(i) => I(i*2)
            | _ => raise TypeError

fun neg (v:value):value =
  case (v) of B(b) => B(not b)
            | _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Example: Typed Expressions

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of value

fun mkI (i:int):IT sva
fun mkB (b:bool):BT sval

fun sdouble (v:IT sva
  S(double (Sof (v))
fun sneg (v:BT svalue
  S(neg (Sof (v)))

fun stoString (v:α svalue
  S(toString (Sof (v)))
```

```
datatype value =
  I of int | B of bool

                    lue):value =
                      :*2)
                    Error

                      g(i)
                    .toString(b)
```

IT and BT : tags for integers and Booleans

IT svalue : type of an int value

BT svalue : type of a bool value

Example: Typed Expressions

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of value

fun mkI (i:int):IT svalue =
fun mkB (b:bool):BT svalue =

fun sdouble (v:IT svalue):
  S(double (Sof (v)))
fun sneg (v:BT svalue):BT sva
  S(neg (Sof (v)))

fun stoString (v:α svalue):string =
  S(toString (Sof (v)))
```

```
datatype value =
  I of int | B of bool

fun double (v:value):value =
                    i*2)
                    rror

                    Error

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

A "safe" value is just a value with a superfluous type variable to hold the tag

A phantom type

Example: Typed Expressions

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of value

fun mkI (i:int):IT svalue = S(I(i
fun mkB (b:bool):BT svalue = S(B

fun sdouble (v:IT svalue):IT sv
  S(double (Sof (v)))
fun sneg (v:BT svalue):BT svalue
  S(neg (Sof (v)))

fun stoString (v:α svalue):string =
  S(toString (Sof (v)))
```

```
datatype value =
  I of int | B of bool
                                  or
                                  ror
                                  ing =
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

All we care about is that

IT ≠ BT

Natural but misleading choice:

IT = int

BT = bool

Example: Typed Expressions

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of value

fun mkI (i:int):IT svalue =
fun mkB (b:bool):BT svalue =

fun sdouble (v:IT svalue):
  S(double (Sof (v)))
fun sneg (v:BT svalue):BT sva
  S(neg (Sof (v)))

fun stoString (v:α svalue):string =
  S(toString (Sof (v)))
```

```
datatype value =
  I of int | B of bool

fun double (v:value):value =
                    i*2)
                    rror

                    Error

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Property needed:

$x \neq y$ iff $x\ \text{svalue} \neq y\ \text{svalue}$

Example: Typed

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of valu

fun mkI (i:int):IT svalue = S(I
fun mkB (b:bool):BT svalue =
```

sdouble (mkB true)

does not type check because BT svalue and IT svalue do not unify

```
                                                    ypeError

fun sdouble (v:IT svalue):IT svalue =
  S(double (Sof (v)))
fun sneg (v:BT svalue):BT svalue =
  S(neg (Sof (v)))

fun stoString (v:α svalue):string =
  S(toString (Sof (v)))
```

```
fun neg (v:value):value =
  case (v) of B(b) => B(not b)
            | _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Example: Typed Expressions

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of valu

fun mkI (i:int):IT svalue =
fun mkB (b:bool):BT svalue =

fun sdouble (v:IT svalue):IT
  S(double (Sof (v)))
fun sneg (v:BT svalue):BT svalue =
  S(neg (Sof (v)))

fun stoString (v:α svalue):string =
  S(toString (Sof (v)))
```

```
                                  => B(not b)
                | _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

sneg (mkI 3)

does not type check because IT svalue and BT svalue do not unify

Example: Typed Expressions

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of value

fun mkI (i:int):IT svalue = S(
fun mkB (b:bool):BT svalue =

fun sdouble (v:IT svalue):IT
  S(double (Sof (v)))
fun sneg (v:BT svalue):BT svalu
  S(neg (Sof (v)))

fun stoString (v:α svalue):string =
  S(toString (Sof (v)))
```

```
datatype value =
  I of int | B of bool
                                  rror
fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

stoString (mkB true)

type checks because
BT svalue and α svalue unify

Example: Typed Expressions

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of value

fun mkI (i:int):IT svalue = S(
fun mkB (b:bool):BT svalue =

fun sdouble (v:IT svalue):IT
  S(double (Sof (v)))
fun sneg (v:BT svalue):BT svalu
  S(neg (Sof (v)))

fun stoString (v:α svalue):string =
  S(toString (Sof (v)))
```

```
datatype value =
  I of int | B of bool
                                  rror

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

stoString (mkI 3)

type checks because
IT svalue and α svalue unify

Example: Typed Expressions

```
datatype IT = Dummy1 of unit
datatype BT = Dummy2 of unit
datatype α svalue = S of value

fun mkI (i:int):IT svalue = S(I(i))
fun mkB (b:bool):BT svalue = S(B(b))

fun sdouble (v:IT svalue):IT svalue =
  S(double (Sof (v)))
fun sneg (v:BT svalue):BT svalue =
  S(neg (Sof (v)))

fun stoString (v:α svalue):string =
  S(toString (Sof (v)))
```

```
datatype value =
  I of int | B of bool

fun double (v:value):value =
  case (v) of I(i) => I(i*2)
            | _ => raise TypeError

fun neg (v:value):value =
  case (v) of B(b) => B(not b)
            | _ => raise TypeError

fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
```

Son, all this S wrapping and unwrapping is bothering me...

Look, Ma, No Wrapping

```
structure SafeValue :> sig
  type IT
  type BT
  type α value
  val mkI: int -> IT value
  val mkB: bool -> BT value
  val double : IT value -> IT value
  val neg : BT value -> BT value
  val toString : α value -> string
end = struct
  datatype IT = Dummy2 of unit
  datatype BT = Dummy1 of unit
  type α value = Value.value
  fun mkI (i) = Value.I(i)
  fun mkB (b) = Value.B(b)
  fun double (v) = Value.double(v)
  fun neg (v) = Value.neg (v)
  fun toString (v) = Value.toString (v)
end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
              | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
              | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Look, Ma, No Wrapping

```
structure SafeValue :> sig
  type IT
  type BT
  type α value
  val mkI: int -> IT value
  val mkB: bool -> BT value
  val double : IT value -> IT v
  val neg : BT value -> BT valu
  val toString : α value -> strin
end = struct
  datatype IT = Dummy2 of unit
  datatype BT = Dummy1 of unit
  type α value = Value.value
  fun mkI (i) = Value.I(i)
  fun mkB (b) = Value.B(b)
  fun double (v) = Value.double(v)
  fun neg (v) = Value.neg (v)
  fun toString (v) = Value.toString (v)
end
```

```
struct
rror
B(not b)
| _ => raise TError
fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
    | B(b) => Bool.toString(b)
end
```

The “safe” type is now just a type abbreviation for the underlying type

Again, with a phantom type

Look, Ma, No Wrapping

```
structure SafeValue :> sig
  type IT
  type BT
  type α value
  val mkI: int -> IT valu
  val mkB: bool -> BT va
  val double : IT value -
  val neg : BT value -> BT
  val toString : α value -> strin
end = struct
  datatype IT = Dummy2 of unit
  datatype BT = Dummy1 of unit
  type α value = Value.value
  fun mkI (i) = Value.I(i)
  fun mkB (b) = Value.B(b)
  fun double (v) = Value.double(v)
  fun neg (v) = Value.neg (v)
  fun toString (v) = Value.toString (v)
end
```

Opaque signature matching gives required equivalence behavior for "safe" values

```
structure Value = struct
                alue =
                f bool

                :value =
                => I(i*2)
                => raise TError
          (v:value):value =
    case (v) of B(b) => B(not b)
                | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Look, Ma, No

Using the module system also lets us reuse the name of types and operations from the underlying "unsafe" implementation

```
structure SafeValue :> sig
  type IT
  type BT
  type α value
  val mkI: int -> IT value
  val mkB: bool -> BT value
  val double : IT value -> IT value
  val neg : BT value -> BT value
  val toString : α value -> string
end = struct
  datatype IT = Dummy2 of unit
  datatype BT = Dummy1 of unit
  type α value = Value.value
  fun mkI (i) = Value.I(i)
  fun mkB (b) = Value.B(b)
  fun double (v) = Value.double(v)
  fun neg (v) = Value.neg (v)
  fun toString (v) = Value.toString (v)
end
```

```
fun double (v:value):value =
  case (v) of I(i) => I(i*2)
            | _ => raise TError
fun neg (v:value):value =
  case (v) of B(b) => B(not b)
            | _ => raise TError
fun toString (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
end
```

Example: Sockets Library

Phantom types are used in the SML/NJ sockets library

SML/NJ view: a socket is a 32-bit word

OS view: a socket is one of

- UDP socket: used for unstructured sessions
- TCP socket: used for structured sessions
- The socket type is recorded internally by the OS

Can devise a "safe" interface to sockets...

The OS Interface

```
type prim_socket = Word32.word



val makeUDP : string -> prim_socket
val makeTCP : string -> prim_socket

val sendUDP :
  prim_socket * string -> unit
val sendTCP :
  prim_socket * string -> unit

val close : prim_socket -> unit
```

The "Safe" Interface

```
datatype α socket =  S of prim_socket
datatype UDP = U of unit
datatype TCP = T of unit

val mkUDP : string -> UDP socket
val mkTCP : string -> TCP socket

val sendUDP :
  UDP socket * string -> unit
val sendTCP :
  TCP socket * string -> unit

val close : α socket -> unit
```

```
type prim_socket = Word32.word

val makeUDP : string -> prim_socket
val makeTCP : string -> prim_socket

val sendUDP :
  prim_socket * string -> unit
val sendTCP :
  prim_socket * string -> unit

val close : prim_socket -> unit
```

A One-Slide Sum Up

The phantom-types technique:

- Use a superfluous type variable to tag data
- Use type constraints to set and check the tag
- Use polymorphism to be polymorphic in the tag

By constructing tags correctly, can capture subtyping

- There is a recipe for doing this
- Depends on how to encode the subtyping hierarchy

The Examples, Abstractly

An underlying primitive type of data

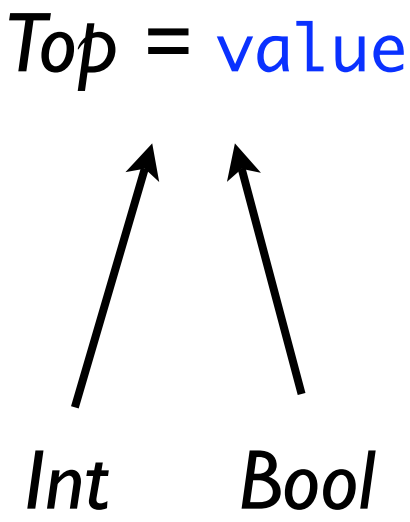
- `value`

Implicit subtypes

- *Int*, *Bool*, *Top*

Operations on the primitive type, constrained at implicit subtypes

- `double` : *Int* → *Int*
- `neg` : *Bool* → *Bool*



The Recipe

Given:

- A primitive type τ_{prim}
- An implicit subtyping hierarchy $\sigma_1, ..., \sigma_n$
- An implementation of τ_{prim} and its operations

Derive:

- A "safe" interface (the types)
- A "safe" implementation (the code)

Want:

- Shared representation and shared operations

Applying the Recipe

Given:

- A primitive type: `value`
- An implicit subtyping hierarchy: *Top*, *Int*, *Bool*
- An implementation: structure `Value`

Derive:

- A "safe" interface: signature `SAFE_VALUE`
- A "safe" implementation: structure `SafeValue`

Deriving the Safe Interface

```
signature SAFE_VALUE = sig

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
              | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
              | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Deriving the Safe Interface

```
signature SAFE_VALUE = sig
  type α value

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
              | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
              | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Deriving the Safe Interface

```
signature SAFE_VALUE = sig
  type α value
  val mkI : int -> <Int>C value
  val mkB : bool -> <Bool>C value

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
              | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
              | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Deriving the Safe Interface

```
signature SAFE_VALUE = sig
  type α value
  val mkI : int -> <Int>C value
  val mkB : bool -> <Bool>C value

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
```

<σ>C is an encoding of σ using some tag

E.g. <*Top*>C = unit
<*Int*>C = int
<*Bool*>C = bool

Deriving the Safe Interface

```
signature SAFE_VALUE = sig
  type α value
  val mkI : int -> <Int>C value
  val mkB : bool -> <Bool>C value
  val double :
      <Int>C value -> <Int>C value

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
              | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
              | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Deriving the Safe Interface

```
signature SAFE_VALUE = sig
  type α value
  val mkI : int -> <Int>C value
  val mkB : bool -> <Bool>C value
  val double :
      <Int>C value -> <Int>C value

  val neg :
      <Bool>C value -> <Bool>C value

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
                | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
                | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Deriving the Safe Interface

```
signature SAFE_VALUE = sig
  type α value
  val mkI : int -> <Int>C value
  val mkB : bool -> <Bool>C value
  val double :
      <Int>C value -> <Int>C value

  val neg :
      <Bool>C value -> <Bool>C value

  val toString :
      <Top>A value -> string

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
                | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
                | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
end
```

Deriving the Safe Interface

```
signature SAFE_VALUE = s
  type α value
  val mkI : int -> <
  val mkB : bool ->
  val double :
      <Int>C value -

  val neg :
      <Bool>C value ->

  val toString :
      <Top>A value -> string

end
```

```
or
ot b)
aise TError
g (v:value):string =
  case (v)
    of I(i) => Int.toString(i)
     | B(b) => Bool.toString(b)
end
```

$\langle\sigma\rangle_A$ is an encoding of σ
and its subtypes

E.g. $\langle Top\rangle_A = \alpha$
$\langle Int\rangle_A$ = int
$\langle Bool\rangle_A$ = bool

Applying the Recipe

Given:

- A primitive type: `value`
- An implicit subtyping hierarchy: *Top*, *Int*, *Bool*
- An implementation: structure `Value`

Derive:

- A "safe" interface: signature `SAFE_VALUE` ✓
- A "safe" implementation: structure `SafeValue`

Deriving a Safe Implementation

```
structure SafeValue :> SAFE_VALUE = struct

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
              | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
              | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Deriving a Safe Implementation

```
structure SafeValue :> SAFE_VALUE = struct
  type α value = Value.value

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
              | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
              | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Deriving a Safe Implementation

```
structure SafeValue :> SAFE_VALUE = struct
  type α value = Value.value
  val mkI = Value.I
  val mkB = Value.B

end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
              | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
              | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Deriving a Safe Implementation

```
structure SafeValue :> SAFE_VALUE = struct
  type α value = Value.value
  val mkI = Value.I
  val mkB = Value.B
  val double = Value.double


  val neg = Value.neg


  val toString = Value.toString


end
```

```
structure Value = struct
  datatype value =
    I of int | B of bool

  fun double (v:value):value =
    case (v) of I(i) => I(i*2)
              | _ => raise TError
  fun neg (v:value):value =
    case (v) of B(b) => B(not b)
              | _ => raise TError
  fun toString (v:value):string =
    case (v)
      of I(i) => Int.toString(i)
       | B(b) => Bool.toString(b)
end
```

Applying the Recipe

Given:

- A primitive type: `value`
- An implicit subtyping hierarchy: *Top*, *Int*, *Bool*
- An implementation: structure `Value`

Derive:

- A "safe" interface: signature `SAFE_VALUE` ✓
- A "safe" implementation: structure `SafeValue` ✓

Subtyping Hierarchies Encodings

The recipe above relies on some encoding satisfying

$$\langle \sigma_1 \rangle_C \text{ unifies with } \langle \sigma_2 \rangle_A \text{ iff } \sigma_1 \leq \sigma_2$$

Our encodings above are ad hoc

We can devise uniform encodings for:

- Tree hierarchies
- Powerset lattice hierarchies

Subtyping Hierarchies Encodings

The recipe above relies on some encoding satisfying

$$\langle \sigma_1 \rangle_C \text{ unifies with } \langle \sigma_2 \rangle_A \text{ iff } \sigma_1 \leq \sigma_2$$

Our encodings above are

We can devise uniform

- Tree hierarchies
- Powerset lattice hierarchi

Unification is symmetric, subtyping is not

So must use two different encodings!

Tree Hierarchies

Let T be a finite tree

Define `datatype α` n_t `= Dummy`$_t$ `of unit`

- One for every node t in T

Encode a node t in T by its path t_1, ... t_k, t:

- $\langle t \rangle_C$ = `unit` n_t n_{tk} `...` n_{t1}
- $\langle t \rangle_A$ = `α` n_t n_{tk} `...` n_{t1}

Tree Hierarchies

Let T be a finite tree

Define `datatype α` n_t `= Dummy`$_t$ `of unit`

- One for every node t in

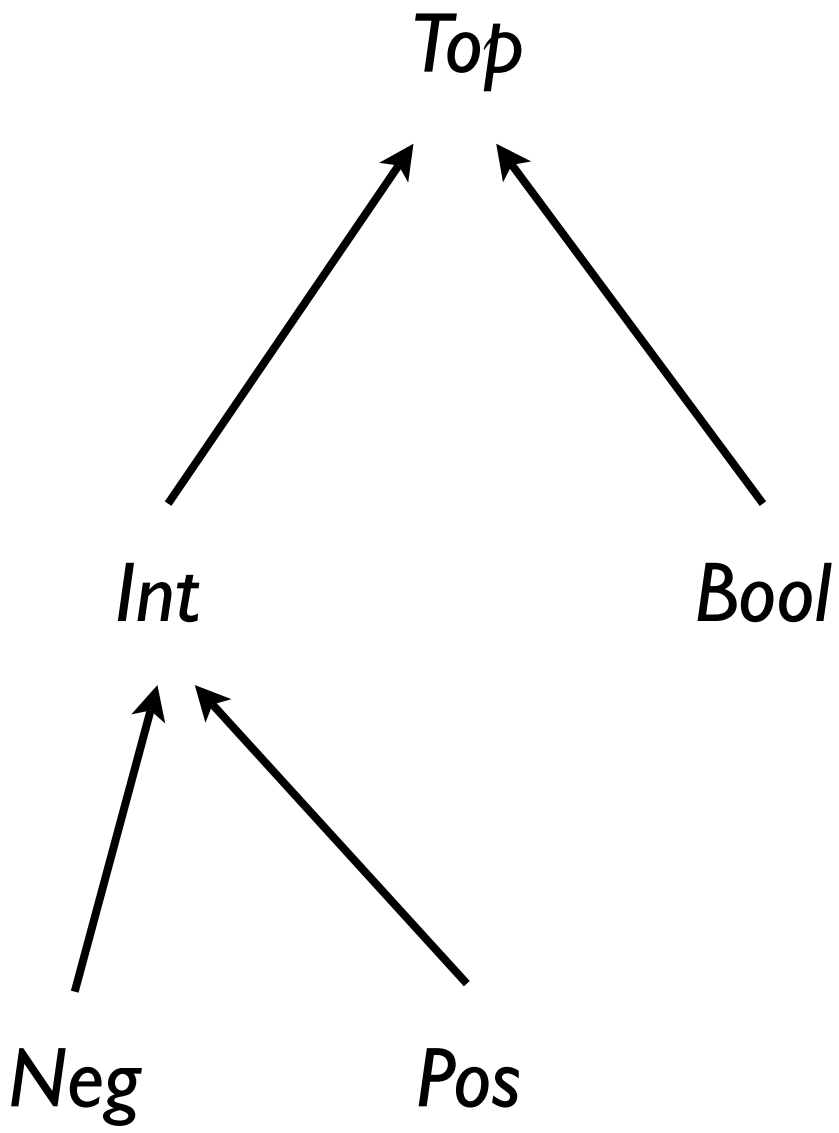
Encode a node t in T by its p

- $\langle t \rangle_C$ = `unit` n_t n_{tk} `...`
- $\langle t \rangle_A$ = `α` n_t n_{tk} `...` n_{t1}

What this is is completely irrelevant

Just need the same equivalence property as before

Example



$\langle Top \rangle_C$ = unit n_{Top}

$\langle Int \rangle_C$ = unit n_{Int} n_{Top}

$\langle Bool \rangle_C$ = unit n_{Bool} n_{Top}

$\langle Neg \rangle_C$ = unit n_{Neg} n_{Int} n_{Top}

$\langle Pos \rangle_C$ = unit n_{Pos} n_{Int} n_{Top}

$\langle Top \rangle_A$ = α n_{Top}

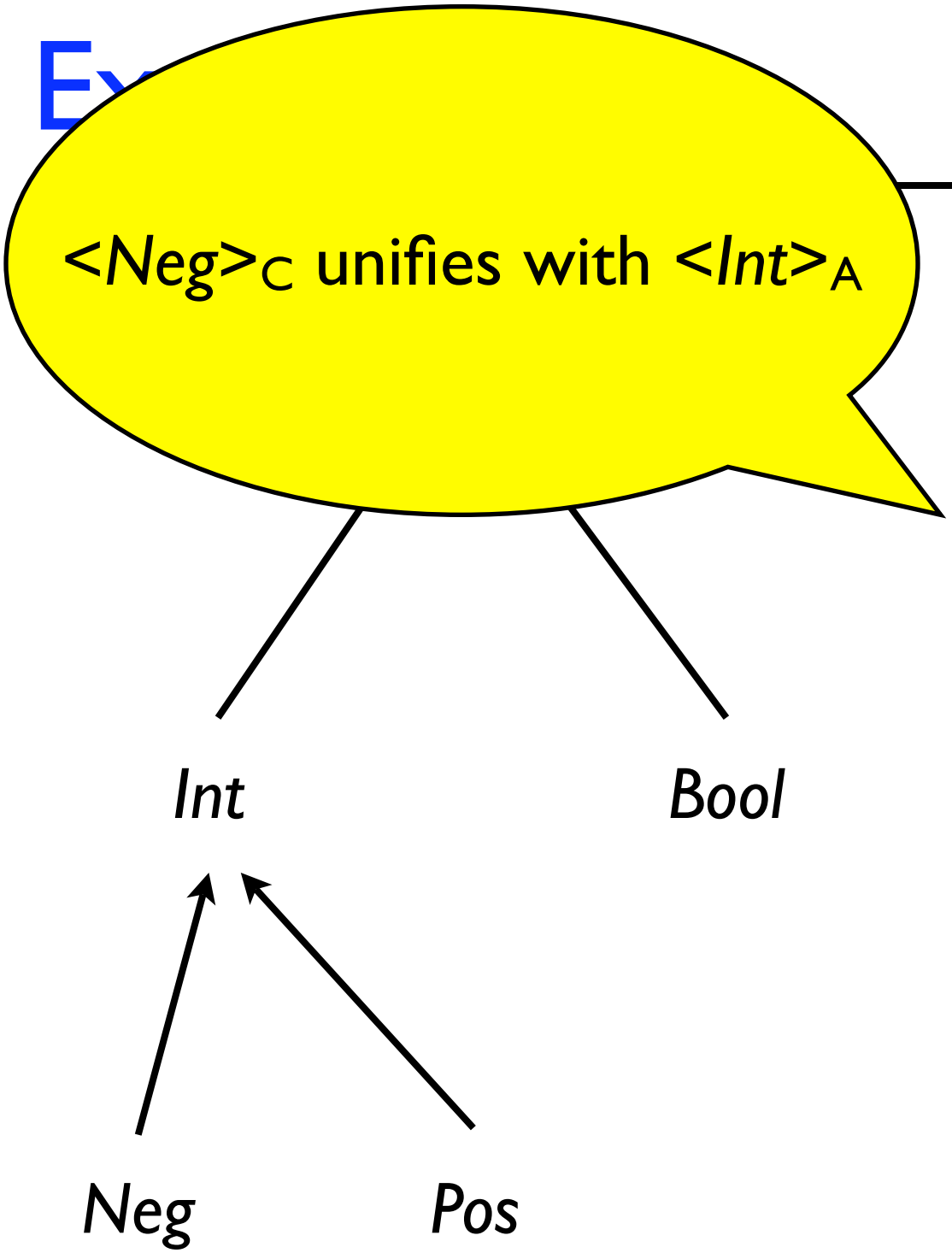
$\langle Int \rangle_A$ = α n_{Int} n_{Top}

$\langle Bool \rangle_A$ = α n_{Bool} n_{Top}

$\langle Neg \rangle_A$ = α n_{Neg} n_{Int} n_{Top}

$\langle Pos \rangle_A$ = α n_{Pos} n_{Int} n_{Top}

Ex



$\langle Top \rangle_C = \texttt{unit}\ n_{Top}$

$\langle Int \rangle_C = \texttt{unit}\ n_{Int}\ n_{Top}$

$\langle Bool \rangle_C = \texttt{unit}\ n_{Bool}\ n_{Top}$

$\langle Neg \rangle_C = \texttt{unit}\ n_{Neg}\ n_{Int}\ n_{Top}$

$\langle Pos \rangle_C = \texttt{unit}\ n_{Pos}\ n_{Int}\ n_{Top}$

$\langle Top \rangle_A = \alpha\ n_{Top}$

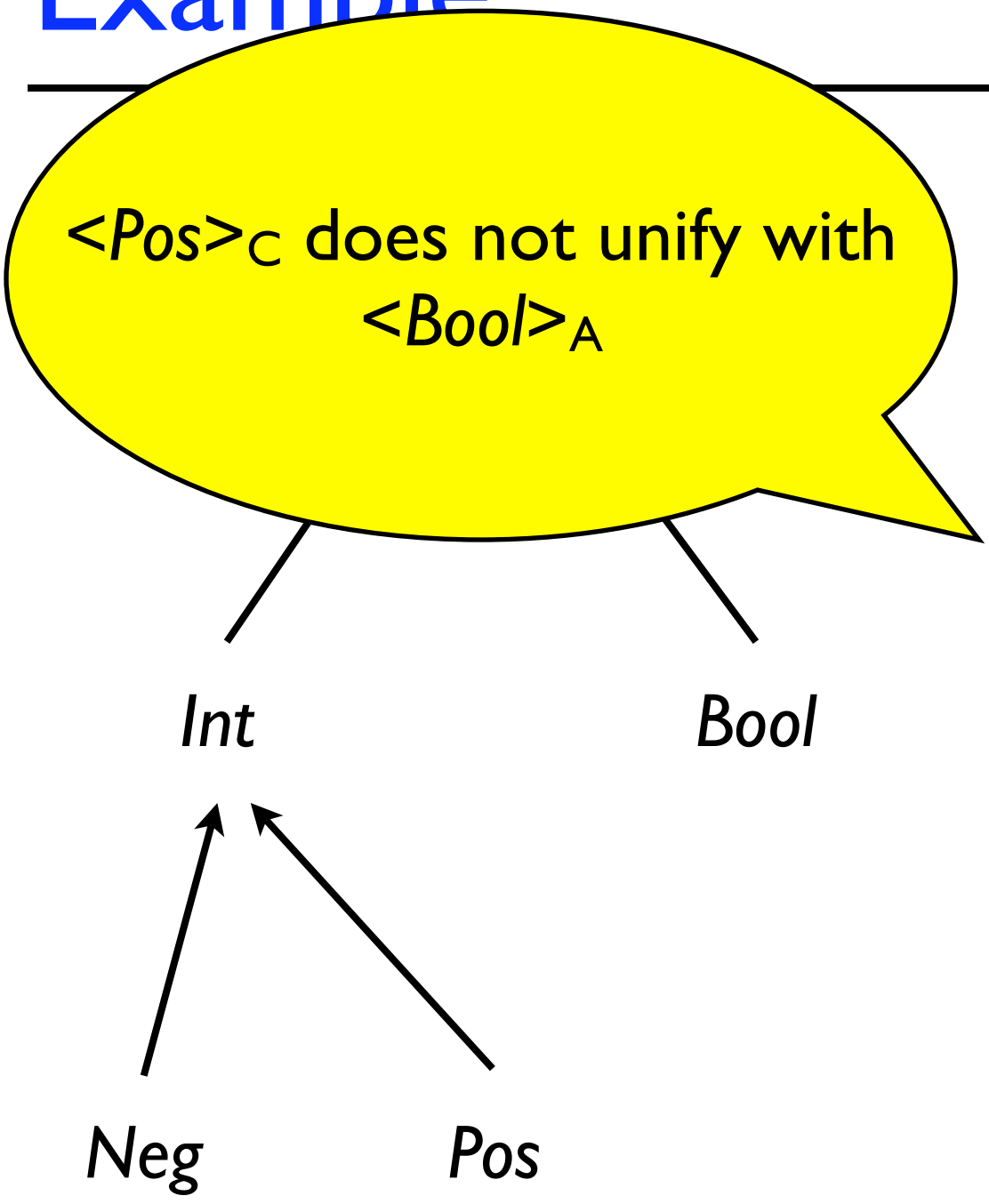
$\langle Int \rangle_A = \alpha\ n_{Int}\ n_{Top}$

$\langle Bool \rangle_A = \alpha\ n_{Bool}\ n_{Top}$

$\langle Neg \rangle_A = \alpha\ n_{Neg}\ n_{Int}\ n_{Top}$

$\langle Pos \rangle_A = \alpha\ n_{Pos}\ n_{Int}\ n_{Top}$

Example



$\langle Top \rangle_C = \mathtt{unit}\ n_{Top}$

$\langle Int \rangle_C = \mathtt{unit}\ n_{Int}\ n_{Top}$

$\langle Bool \rangle_C = \mathtt{unit}\ n_{Bool}\ n_{Top}$

$\langle Neg \rangle_C = \mathtt{unit}\ n_{Neg}\ n_{Int}\ n_{Top}$

$\langle Pos \rangle_C = \mathtt{unit}\ n_{Pos}\ n_{Int}\ n_{Top}$

$\langle Top \rangle_A = \alpha\ n_{Top}$

$\langle Int \rangle_A = \alpha\ n_{Int}\ n_{Top}$

$\langle Bool \rangle_A = \alpha\ n_{Bool}\ n_{Top}$

$\langle Neg \rangle_A = \alpha\ n_{Neg}\ n_{Int}\ n_{Top}$

$\langle Pos \rangle_A = \alpha\ n_{Pos}\ n_{Int}\ n_{Top}$

Powerset Lattice Hierarchies

Let $S = \{s_1,\ldots,s_n\}$ be a finite set

- L = set of subsets of S ordered by inclusion

Define `datatype α z = Dummy of unit`

Given $X \subseteq S$:

$$\langle X\rangle_C = t_1 \texttt{ * } \ldots \texttt{ * } t_n \text{ where } t_i = \begin{cases} \texttt{unit} & \text{if } s_i \in X \\ \texttt{unit z} & \text{o/w} \end{cases}$$

$$\langle X\rangle_A = t_1 \texttt{ * } \ldots \texttt{ * } t_n \text{ where } t_i = \begin{cases} \alpha_i & \text{if } s_i \in X \\ \alpha_i\ \texttt{z} & \text{o/w} \end{cases}$$

Powerset Lattice Hierarchies

Let S = {s_1,...,s_n} [illegible] set

- L = [illegible] ed by inclusion

Think of `unit` as ON and `unit z` as OFF

Define [illegible] `nit`

Given X ⊆ [illegible]

$\langle X \rangle_C = t_1$ `*` ... `*` t_n where t_i = `unit` if $s_i \in X$

`unit z` o/w

$\langle X \rangle_A = t_1$ `*` ... `*` t_n where t_i = α_i if $s_i \in X$

α_i `z` o/w

Powerset Lattice Hierarchies

Let $S = \{s_1,\ldots,s_n\}$ be a finite set

- L = set of subsets of S ordered by inclusion

Define `datatype α z = Dummy of unit`

Given $X \subseteq S$:

$\langle X \rangle_C = t_1$ `*` $\ldots$ `*` t_n where

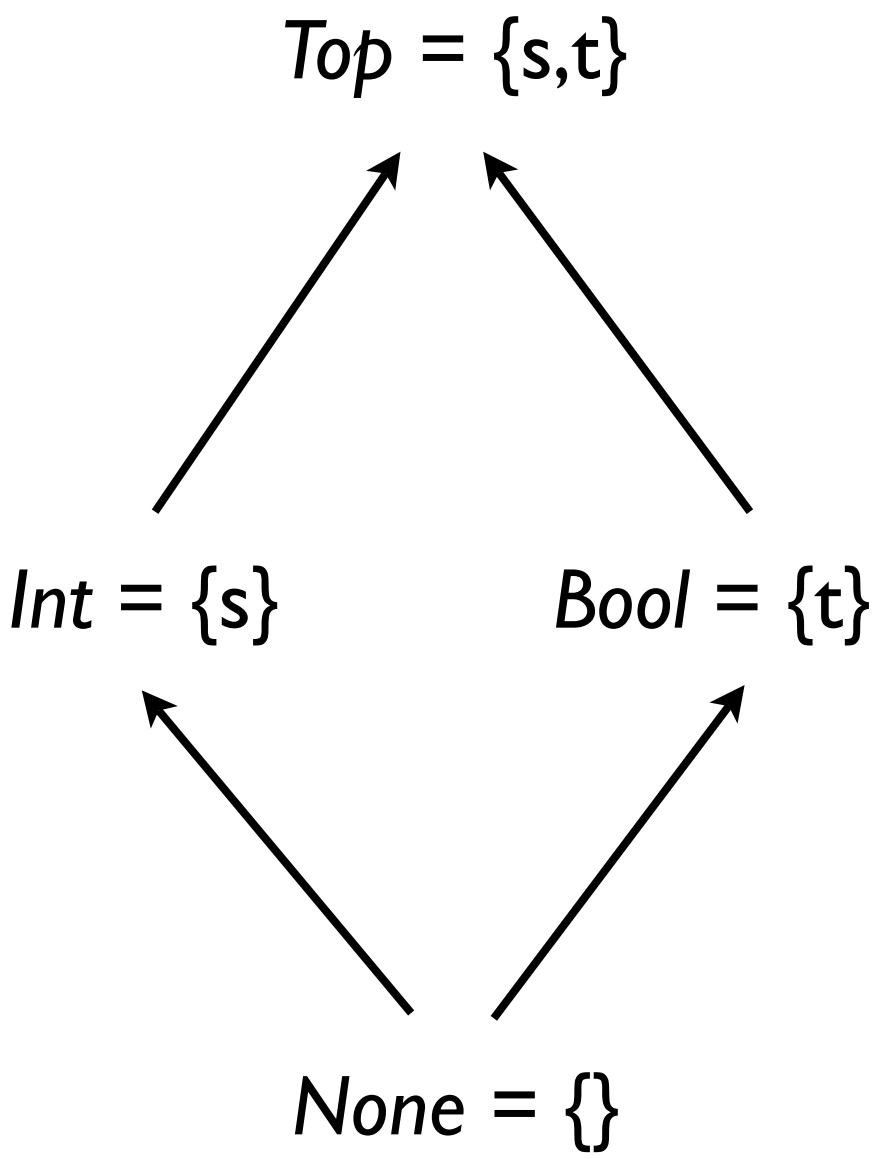
`unit z` o/w

$\langle X \rangle_A = t_1$ `*` $\ldots$ `*` t_n where $t_i =$ `αi` if $s_i \in X$

`αi z` o/w

Again, what this is is completely irrelevant

Example



$\langle Top \rangle_C$ = `unit * unit`

$\langle Int \rangle_C$ = `unit * unit z`

$\langle Bool \rangle_C$ = `unit z * unit`

$\langle None \rangle_C$ = `unit z * unit z`

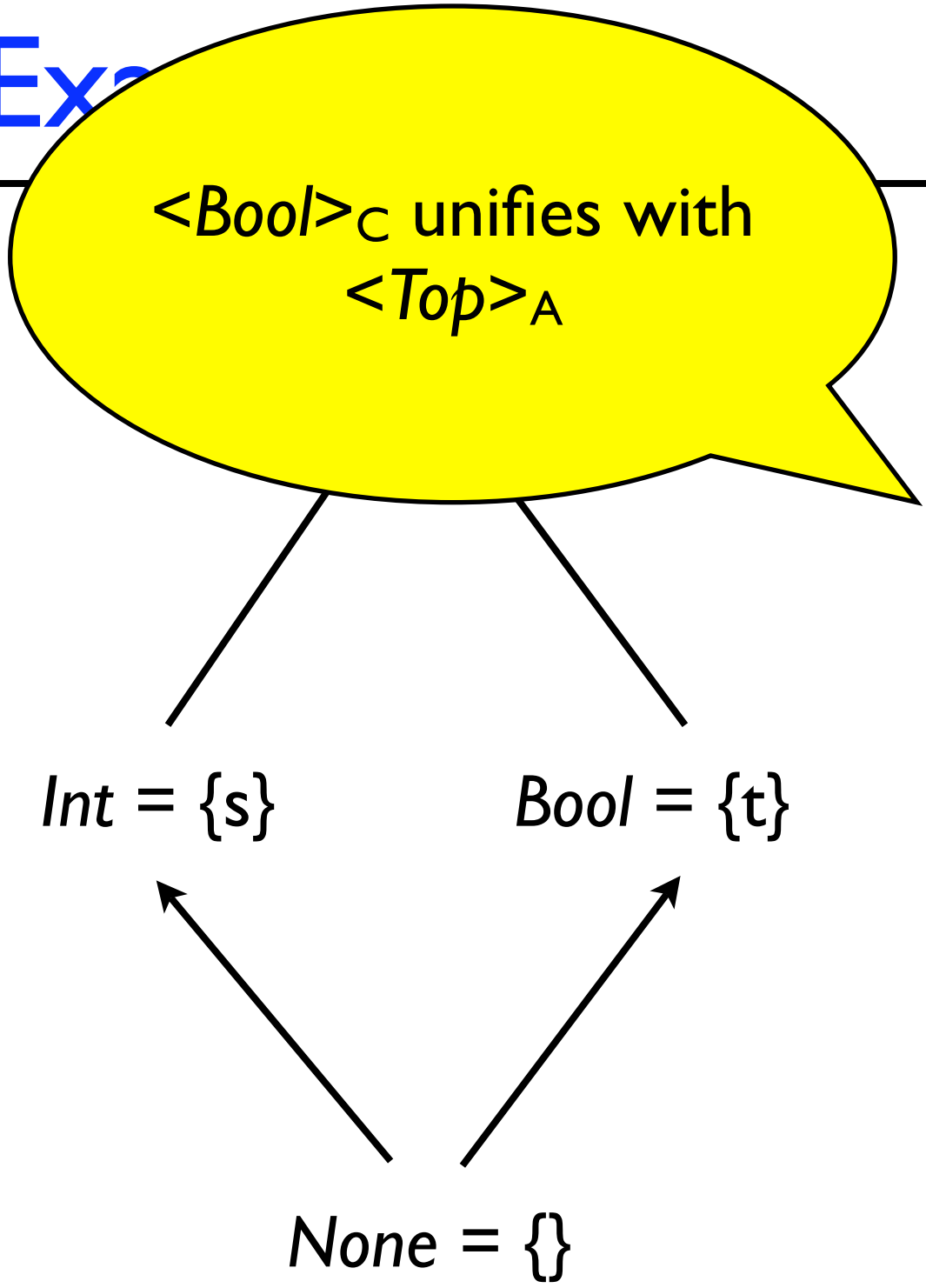
$\langle Top \rangle_A$ = α_1 * α_2

$\langle Int \rangle_A$ = α_1 * α_2 z

$\langle Bool \rangle_A$ = α_1 z * α_2

$\langle None \rangle_A$ = α_1 z * α_2 z

Ex



$\langle Top \rangle_C$ = `unit * unit`

$\langle Int \rangle_C$ = `unit * unit z`

$\langle Bool \rangle_C$ = `unit z * unit`

$\langle None \rangle_C$ = `unit z * unit z`

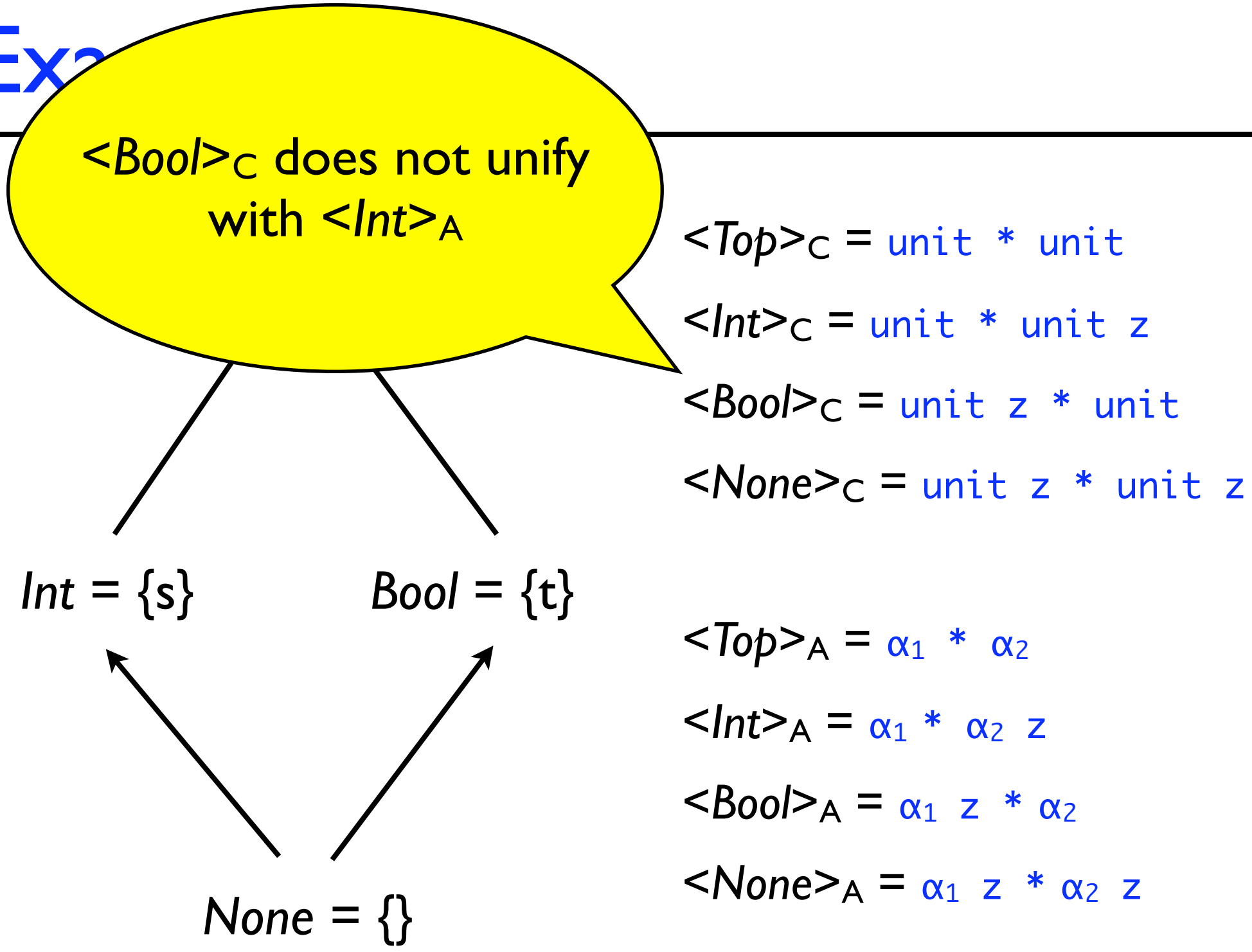
$\langle Top \rangle_A$ = α_1 * α_2

$\langle Int \rangle_A$ = α_1 * α_2 z

$\langle Bool \rangle_A$ = α_1 z * α_2

$\langle None \rangle_A$ = α_1 z * α_2 z

Ex


<Bool>C does not unify with <Int>A
<Top>C = unit * unit
<Int>C = unit * unit z
<Bool>C = unit z * unit
<None>C = unit z * unit z
Int = {s}
Bool = {t}
None = {}
<Top>A = α1 * α2
<Int>A = α1 * α2 z
<Bool>A = α1 z * α2
<None>A = α1 z * α2 z

Other Results

Can encode sublattices of powerset lattices

- Thus, any finite partially ordered hierarchy

Can define extensible encodings *under some conditions*

- How to add subtypes to a hierarchy without recomputing the encoding

Other Results

TO SUM UP

Can encode sublattices of powerset lattices

- Thus, any finite partially ordered hierarchy

(1)

Choose encoding for implicit subtype hierarchy

Can define extensible encodings *under some conditions*

- How to add subtypes to a hierarchy without recompiling the encoding

(2)

Create safe interface following recipe

A Form of Type Safety

We can prove that the recipe produces a "safe" interface, roughly as follows:

- $\lambda^{\leq}$: a simply typed lambda calculus with subtyping only on implicit types
- λ^{DM} : a simply typed lambda calculus with a Hindley-Milner type system and datatypes
- The recipe is a translation T from $\lambda^{\leq}$ to λ^{DM}

Theorem: If $\vdash e : \tau$ in $\lambda^{\leq}$, then $\vdash T[e] : T[\tau]$ in λ^{DM}

Conclusions

Phantom types are part of the folklore of typed FP

- Uses are simple instances of (first-order) subtyping
- Shown we can capture any subtyping hierarchy
- Proved formally the safety of the approach

Easy extension of the recipe deals with a restricted form of bounded subtyping

- E.g., $\forall \tau \leq \tau'.\tau \rightarrow \tau$

Open question: expressive power of phantom types