Parameterized Modules for Classes and Extensible Functions

/ke • no͞o/
Keunwoo Lee
computational nihilist (grad student)

Craig Chambers
(advisor)

University of Washington

FOOL/WOOD workshop, Jan. 2006

Consider a module...

```
module Lang = {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

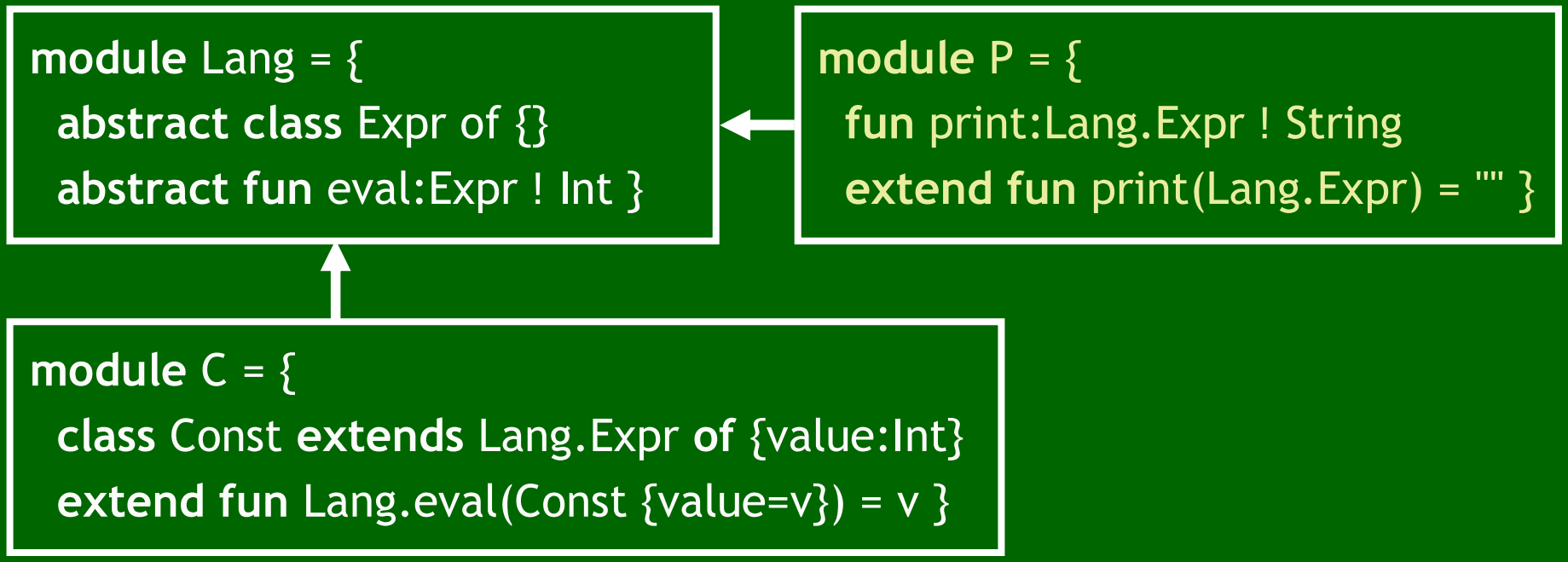
Desideratum 1:
Data type extensibility

```
module Lang = {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```



```
module C = {
  class Const extends Lang.Expr of {value:Int}
  extend fun Lang.eval(Const {value=v}) = v }
```

Desideratum 2:
Function extensibility



Desideratum 3:
Reusable extensions

```
module Lang = {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

```
signature LangSig = sig {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

```
module Plus = MakePlus(Lang) (* {
  class Plus extends Lang.Expr
     of {left:Lang.Expr, right:Lang.Expr}
  extend fun Lang.eval(Plus ...) ... } *)
```

```
module MakePlus = (L:LangSig) ! {
  class Plus extends L.Expr
     of {left:L.Expr, right:L.Expr}
  extend fun L.eval(Plus {left, right}) = ... }
```

Desideratum 4:
Modular typechecking

```
module Lang = {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

```
: sig {
    abstract class Expr of {}
    abstract fun eval:Expr ! Int }
```

```
signature LangSig = sig {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

```
module Plus = MakePlus(Lang)
```

```
module MakePlus = (L:LangSig) ! {
  class Plus extends L.Expr
      of {left:L.Expr, right:L.Expr}
  extend fun L.eval(Plus {left, right}) = ... }
```

Desiderata summary

1. Data type extensibility
2. Function extensibility
3. Reusable extensions
4. Modular typechecking

Related work

- ML module system [MacQueen 84, …, Dreyer+03]
  - No nontrivial extensible datatypes or functions
- OO module systems: Jiazzi [McDirmid+01], JavaMod [Ancona/Zucca01], etc.
  - Single-dispatch; function extensibility via design pattern
- OO parameterized classes:
  - Multiple inheritance, mixins [Bracha/Cook90], traits [Schärli+03,etc.]
  - Virtual types [Madsen/Møller-Pedersen89]: gbeta [Ernst01], Scala [Odersky+04], etc.
  - Single-dispatch; function extensibility via design pattern

Previous work: EML [Millstein et al., ICFP'02]

- Extensible datatypes/functions, modular typechecking
- A simple, restrictive parameterized module system

- Present work: **F(EML)**
  - extension of EML with much more useful functors

Outline

- Desiderata
- **Signatures: key issues**
- Solutions
- Conclusions

MakePlus, revisited

```
module Lang = {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

```
: sig {
    abstract class Expr of {}
    abstract fun eval:Expr ! Int }
```

```
signature LangSig = sig {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

```
module Plus = MakePlus(Lang)
```

```
module MakePlus = (L:LangSig) ! {
  class Plus extends L.Expr
      of {left:L.Expr, right:L.Expr}
  extend fun L.eval(Plus {left, right}) = ... }
```

A straw man signature calculus

Signatures are compatible iff:

- Exact syntactic match on declarations
- Width subtyping: freely permit subsumption to "forget" extra declarations

Both inflexible and unsound

Signatures: Key issues

Signatures constrain:

- Names and modularization choices) **alias declarations**

```
module Lang' = {
  alias class Expr = Foo.Exp
  alias fun eval = Bar.eval }
```

```
sig {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

▪

```
signature LangSig = sig {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

Signatures: Key issues

Signatures constrain:

- Names and modularization choices
- Relationships among declarations/types

) **richer relation language**

```
sig {
  abstract class Expr
     extends Object of {}
  abstract fun eval:Expr ! Int }
```

▪

```
signature LangSig = sig {
  abstract class Expr
     < Object of {}
  abstract fun eval:Expr ! Int }
```

Signatures: Key issues

Signatures constrain:

- Names and modularization choices
- Relationships among declarations/types

```
sig {
  abstract class Expr
     extends Object of {}
  abstract fun eval:Expr ! Int
  abstract fun f:Expr ! String }
```

?

.

```
signature LangSig = sig {
  abstract class Expr
     extends Object of {}
  abstract fun eval:Expr ! Int }
```

- Extra parts that may be ignored

) **width subtyping via sealing**

Signatures: Key issues

Signatures constrain:

- Names and modularization choices) **alias declarations**
- Relationships among declarations/types) **richer relation language**

- Extra parts that may be ignored) **width subtyping via sealing**

Outline

- Desiderata
- Signatures: key issues
- **Solutions**
  - Enriched relations
  - Sealing for subsumption
- Conclusions

Class relations

```
module Lang = {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

```
signature LangSig = sig {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

```
module MakePlus = (L:LangSig) ! {
  class Plus extends L.Expr
      of {left:L.Expr, right:L.Expr}
  extend fun L.eval(Plus {left, right}) = … }
```

Class relations

```
module Lang = {
  abstract class Expr of {}
  abstract fun eval:Expr ! Int }
```

```
module B uses Lang = {
  abstract class BinOp
      extends Lang.Expr of ... }
```

```
module Comm uses B, Lang = {
  abstract class BinOp
      extends B.BinOp of ... }
```

```
signature BOSig = sig {
  abstract class BinOp
      extends Lang.Expr of ... }
```

```
module MakePlus = (O:BOSig) ! {
  class Plus extends O.BinOp of ...
  extend fun Lang.eval(Plus ...) = ... }
```

Enriched class relations

$C \cdot C'$	subtyping
$C < C'$	strict subtyping
$C <^{k} C'$	k-level subtyping
	$C <^{1} C' \acute{} C$ **extends** C'
	$C <^{0} C' \acute{} C = C'$
$C \neq C'$	non-aliasing
C **disjoint** C'	non-intersection
fresh C	freshly declared class

Why these relations?

- Languages with symmetric-best-match dispatch care about *inequalities* and *non-intersection* (which is unusual)

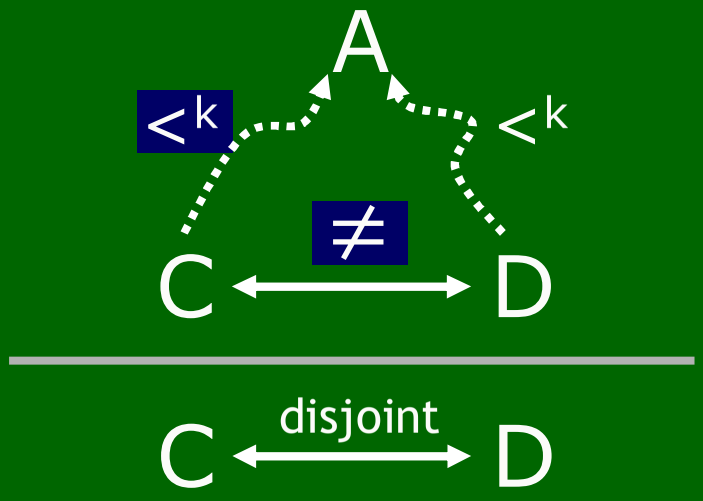
- How does one prove methods unambiguous?

  fun f:C ! Unit
  extend fun f(C) = ()
  extend fun f(D) = ()

  - C Å D is non-empty, computable, and covered by a case of f (requires ·)
  - D < C
  - C **disjoint** D

Deriving disjointness

$$\frac{\mathcal{K} \vdash C_1 \neq C_2 \qquad \mathcal{K} \vdash C_1 <^k C \qquad \mathcal{K} \vdash C_2 <^k C}{\mathcal{K} \vdash C_1 \not\sqcap C_2}\ (\text{CRel-Dis})$$



Deriving distinctness

- How to *modularly* deduce non-aliasing?

```
module C = {
  class Const extends Lang.Expr ... }
```

```
: sig { class Const ...
        fresh Const
        where Const <1 Lang.Expr }
```

```
module N = {
  class Neg extends Lang.Expr ... }
```

```
: sig { class Neg of {...}
        fresh Neg
        where Neg <1 Lang.Expr }
```

```
module X = (A:sig {
      class Neg
      where Neg disjoint C.Const, ...
  }) = ...
```

```
module NBad = {
  alias class Neg = C.Const ... }
```

Enriched class relations

$C \cdot C'$	subtyping
$C < C'$	strict subtyping
$C <^{k} C'$	k-level subtyping
	$C <^{1} C' \acute{} C$ **extends** C'
	$C <^{0} C' \acute{} C = C'$
$C \neq C'$	non-aliasing
C **disjoint** C'	non-intersection
fresh C	freshly declared class

Outline

- Desiderata
- Signatures: key issues
- **Solutions**
  - Enriched relations
  - Sealing for subsumption
- Conclusions

Hiding methods: naïve width subtyping fails

```
module M = {
  fun f:(Object, Object) ! Unit
  extend fun f(Object, Object) = ()
  extend fun f(Object, Int) = () }
```

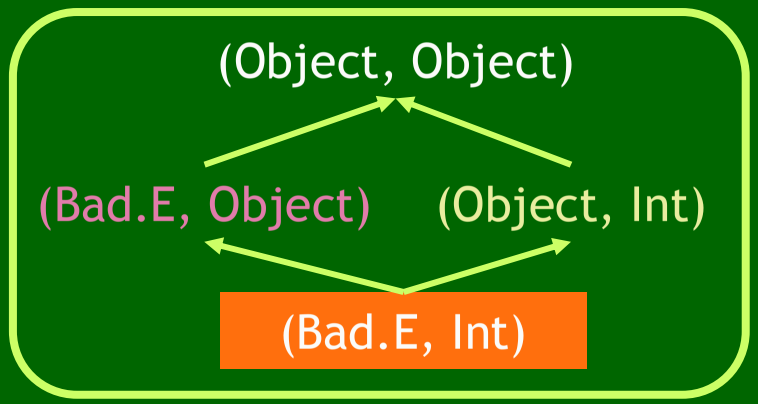
:

```
sig {
  fun f:(Object, Object) ! Unit
  extend fun f(Object, Object)
  extend fun f(Object, Int) }
```

?

```
signature S = sig {
  fun f:(Object, Object) ! Unit }
```

```
module BreakMS = (A:S) ! {
  class E extends Object
  extend fun A.f(E, Object) = () }
```




```
module Bad = BreakMS(M)  (* {
  class E extends Object
  extend fun M.f(E, Object) = ()  } *)
```

What's going on?

- A signature's visible parts give clients both
  capabilities to extend, and
  obligations they incur by so extending

- Must never *hide* an obligation while *granting* the associated capability…

What's going on?

```
module M = {
  fun f:(Object, Object) ! Unit
  extend fun f(Object, Object) = ()
  extend fun f(Object, Int) = () }
```

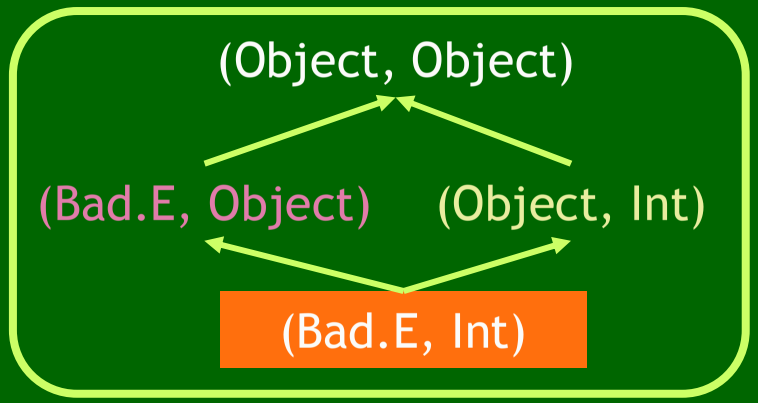
:

```
sig {
  fun f:(Object, Object) ! Unit
  extend fun f(Object, Object)
  extend fun f(Object, Int) }
```

?

```
signature S = sig {
  fun f:(Object, Object) ! Unit }
```

Cannot allow both hiding here...




```
module BreakMS = (A:S) ! {
  class E extends Object
  extend fun A.f(E, Object) = () }
```

...*and* extension here

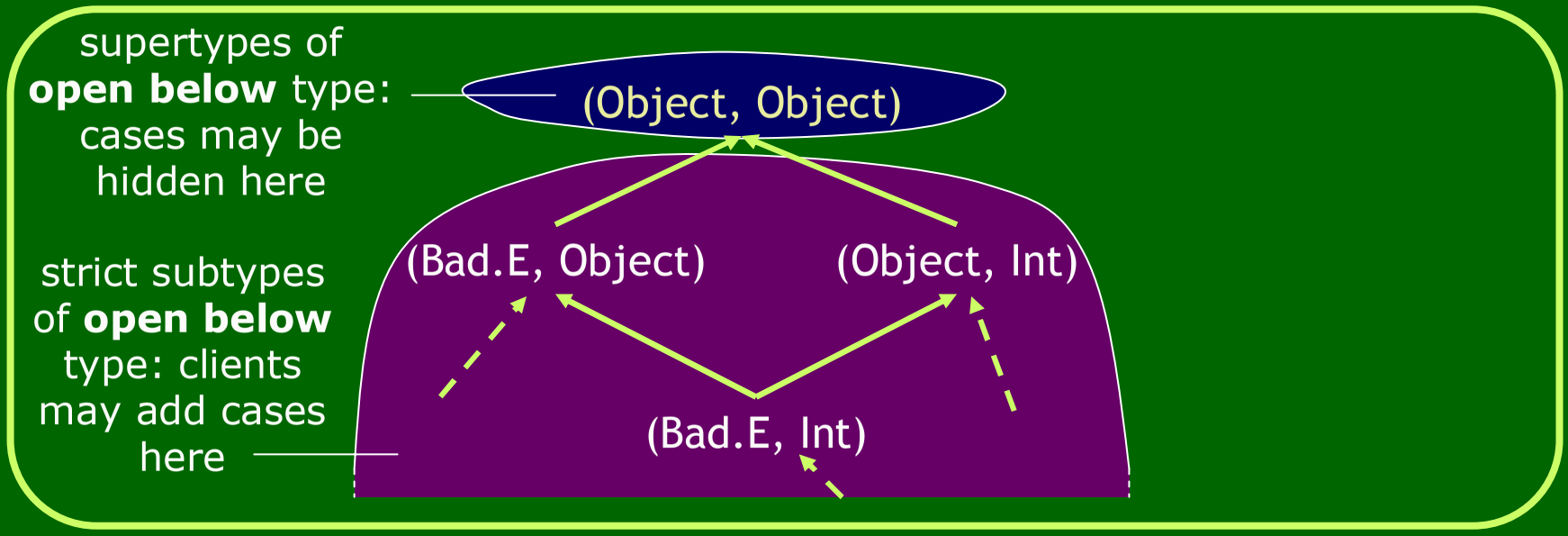
```
module Bad = BreakMS(M) (* {
  class E extends Object
  extend fun M.f(E, Object) = () } *)
```

open below

```
sig {
  fun f:(Object, Object) ! Unit
      open below (Object, Object)
  extend fun f(Object, Object)
  extend fun f(Object, Int) }
```

▪

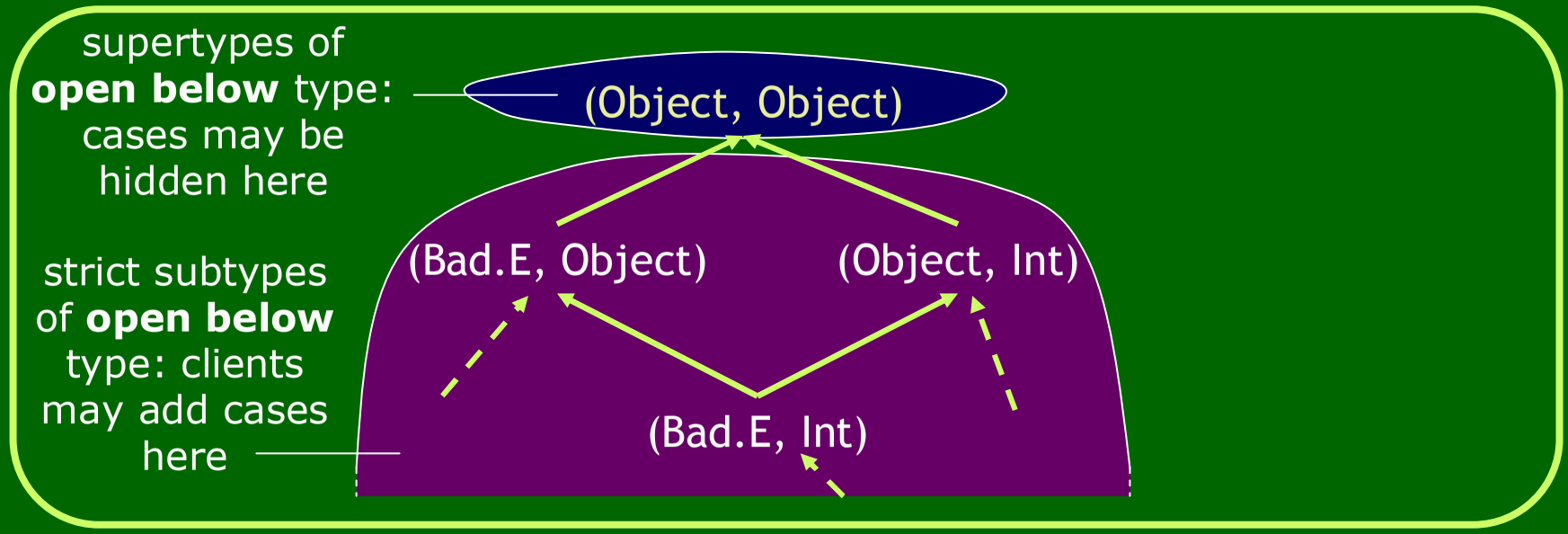
```
sig {
  fun f:(Object, Object) ! Unit
      open below (Object, Object)

  extend fun f(Object, Int) }
```



open below

```
sig {
  fun f:(Object, Object) ! Unit
      open below (Object, Object)
  extend fun f(Object, Object)
  extend fun f(Object, Int) }
```

```
sig {
  fun f:(Object, Object) ! Unit
      open below (Object, Object) }
```

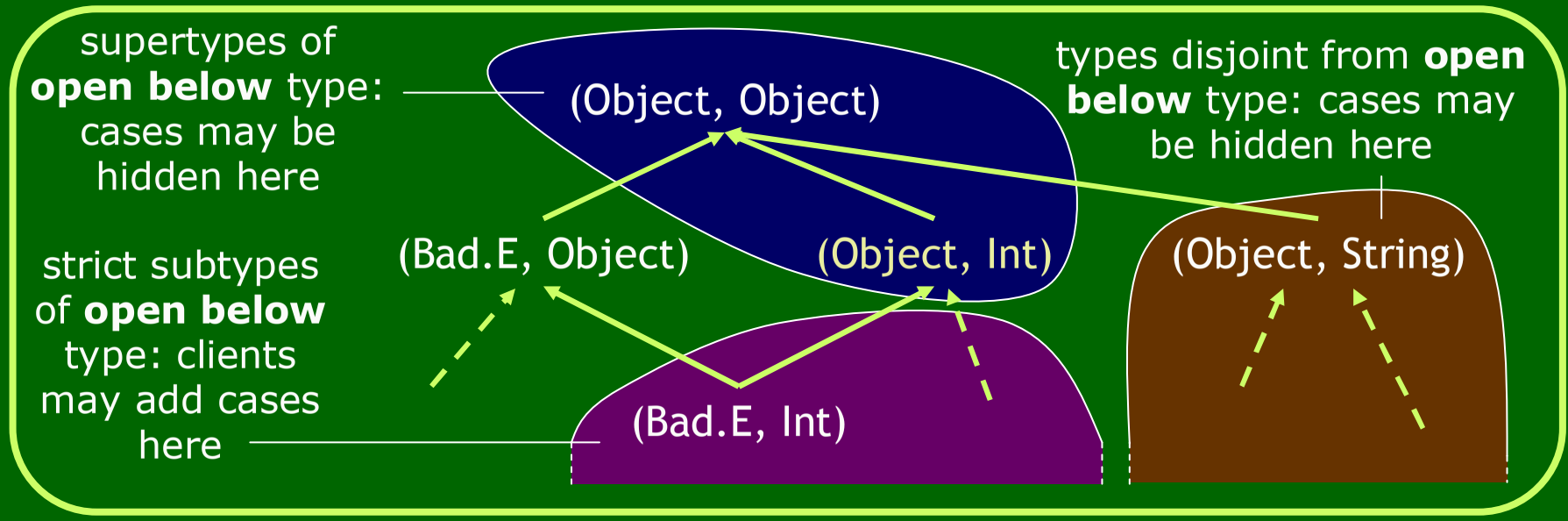


open below

```
sig {
  fun f:(Object, Object) ! Unit
      open below (Object, Int)
  extend fun f(Object, Object)
  extend fun f(Object, Int)
  extend fun f(Object, String) }
```

▪

```
sig {
  fun f:(Object, Object) ! Unit
      open below (Object, Int) }
```

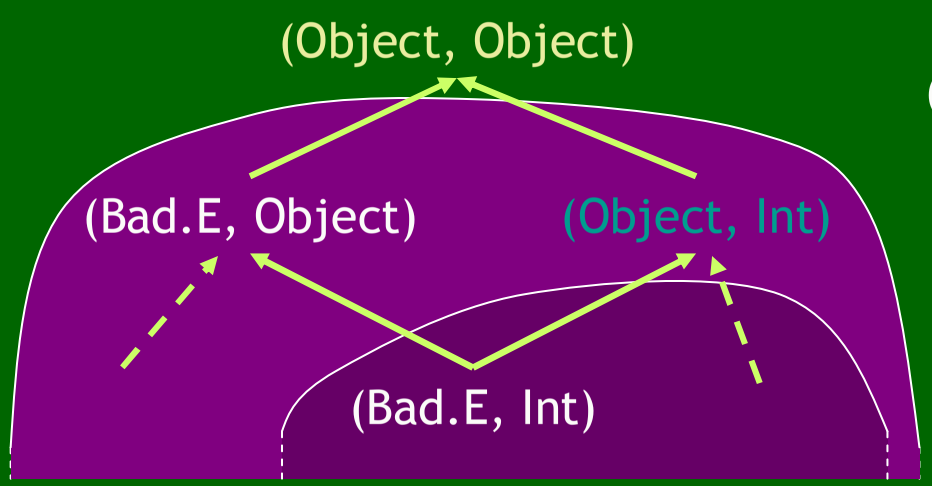


open below

```
sig {
  fun f:(Object, Object) ! Unit
      open below (Object, Object) }
```

▪

```
sig {
  fun f:(Object, Object) ! Unit
      open below (Object, Int) }
```



any client that extends below (Object, Int) also extends below (Object, Object)

Hiding functions

- Unsafe to hide an abstract function, *and* permit concrete extension of class on which that function is abstract
- Class sigs can be **closed**:

  closed class C …

  denotes class that cannot be extended via this signature
- Abstract functions can only be hidden when "owning" class is closed (& non-instantiable)

Outline

- Desiderata
- Signatures: key issues
- Solutions
- **Conclusions**

Status

- Implementation
  - Prototype interpreter: handles our examples
- Formalization
  - Small-step operational semantics
  - Partial soundness proof (via translation to EML)
  - Summary in paper; details/full proof in technical report (forthcoming) and dissertation (forthcoming, somewhat later)

Conclusions

- F(EML) combines extensibility, code reuse, and modular typechecking; key features:
  - Aliases for renaming/remodularization
  - Rich language of declaration/type relations
  - Signature subsumption via selective sealing
- Future work:
  - Hierarchical checking of modularity obligations for nested modules (relaxed checking)
  - Multiple dispatch + virtual types + (mutual) recursion