Advanced Module Systems

(A Guide for the Perplexed)

Benjamin C. Pierce
University of Pennsylvania

Joint work with
Robert Harper (CMU)

Modules: the old is new again

1960s – 70s	modules key technology for “programming in the large”

Modules: the old is new again

1960s – 70s	modules key technology for “programming in the large”
80s – early 90s	objects and classes key technology for “programming”

Modules: the old is new again

1960s – 70s	modules key technology for “programming in the large”
80s – early 90s	objects and classes key technology for “programming” (incorporating many features of modules)

Modules: the old is new again

1960s – 70s	modules key technology for "programming in the large"
80s – early 90s	objects and classes key technology for "programming" (incorporating many features of modules)
mid-90s – ∞	components key technology for "software composition"

Modules: the old is new again

1960s – 70s	modules key technology for “programming in the large”
80s – early 90s	objects and classes key technology for “programming” (incorporating many features of modules)
mid-90s – ∞	components key technology for “software composition” ($\simeq$ modules)

A puzzle

- Recent academic languages (SML, OCaml, MzScheme, etc.) offer complex module features [functors, sharing specifications, H-O / applicative / generative...]

A puzzle

- Recent academic languages (SML, OCaml, MzScheme, etc.) offer complex module features [functors, sharing specifications, H-O / applicative / generative...], plus claims that their features are needed to build large software systems.

A puzzle

- Recent academic languages (SML, OCaml, MzScheme, etc.) offer complex module features [functors, sharing specifications, H-O / applicative / generative...], plus claims that their features are needed to build large software systems.
- Most production languages (C, C++, Java, etc.) provide very simple module systems...

A puzzle

- Recent academic languages (SML, OCaml, MzScheme, etc.) offer complex module features [functors, sharing specifications, H-O / applicative / generative...], plus claims that their features are needed to build large software systems.
- Most production languages (C, C++, Java, etc.) provide very simple module systems... and are believed to “work pretty well” for building large software systems.

A puzzle

- Recent academic languages (SML, OCaml, MzScheme, etc.) offer complex module features [functors, sharing specifications, H-O / applicative / generative...], plus claims that their features are needed to build large software systems.
- Most production languages (C, C++, Java, etc.) provide very simple module systems... and are believed to “work pretty well” for building large software systems.

So: Who is “right”? Or: better question...

What pragmatic issues motivate the features of advanced module systems?

When do we really need which features?

An outsider's tutorial on module systems

What pragmatic issues motivate the features of advanced module systems?

When do we really need which features?

An outsider's tutorial on module systems

What pragmatic issues motivate the features of advanced module systems?

When do we really need which features?

Focus on one particular set of issues:

- specific vs. generic references to external modules
- different ways of managing coherence: sharing by parameterization vs. sharing by specification

Some disclaimers

- Very complex and interconnected set of issues
- Many other (equally tricky) issues omitted
- Difficult to talk clearly about > 1 module system at a time!

Much of the material is familiar; the goal is to organize the “story” so that the choice-points in the design space are as clear as possible.

Lifecycle of a program

(1) development (2) linking (3) execution

A
```
f = λx:int.x+1
g = λx:int.x+2
```

B
```
h = λx:int. A.f(x+3)
```

C
```
i = λx:int. A.g(x)+4
```

D
```
C.i(B.h(5))
```

(including coding, typechecking, compilation)

Lifecycle of a program

(1) development (2) linking (3) execution

Code...

```
A
f = λx:int.x+1
g = λx:int.x+2
```

```
B
h = λx:int. A.f(x+3)
```

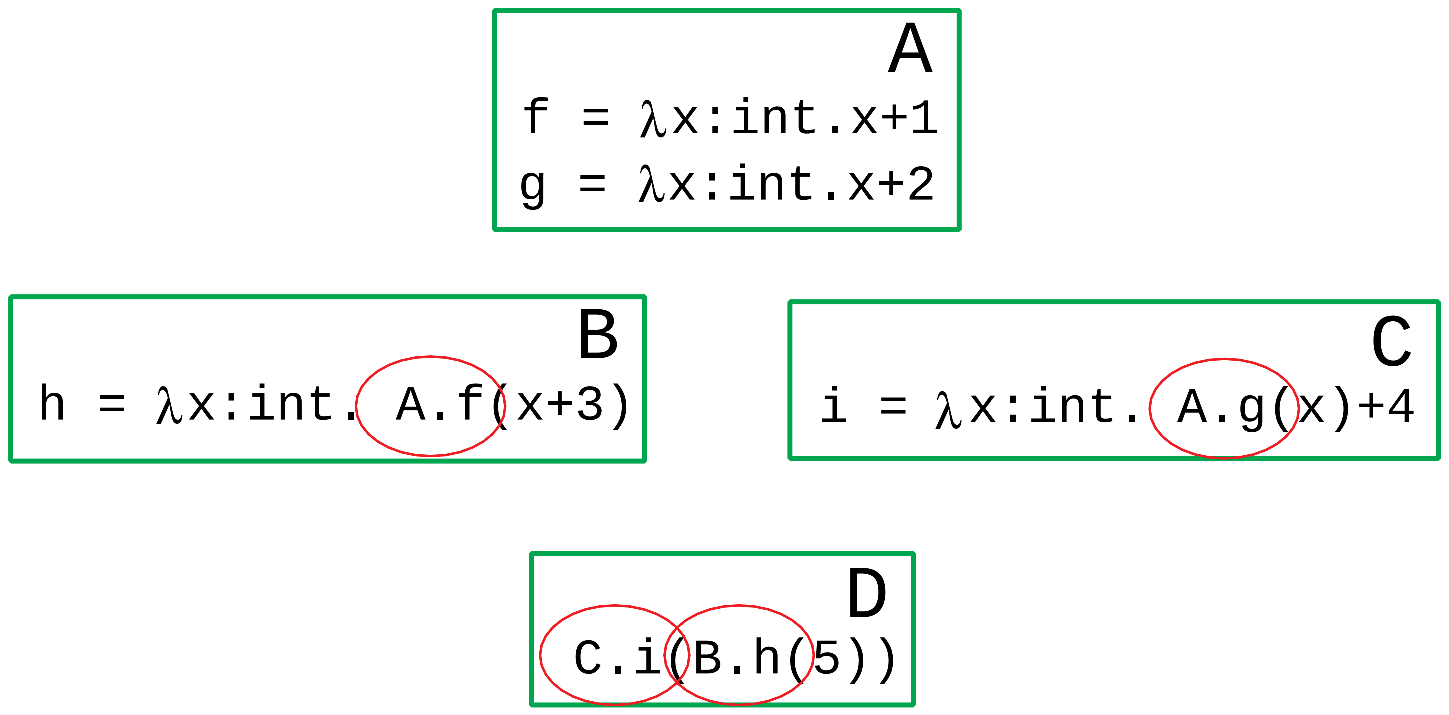
```
C
i = λx:int. A.g(x)+4
```

```
D
C.i(B.h(5))
```

...grouped into modules

Lifecycle of a program

(1) development (2) linking (3) execution



```
A
f = λx:int.x+1
g = λx:int.x+2
```

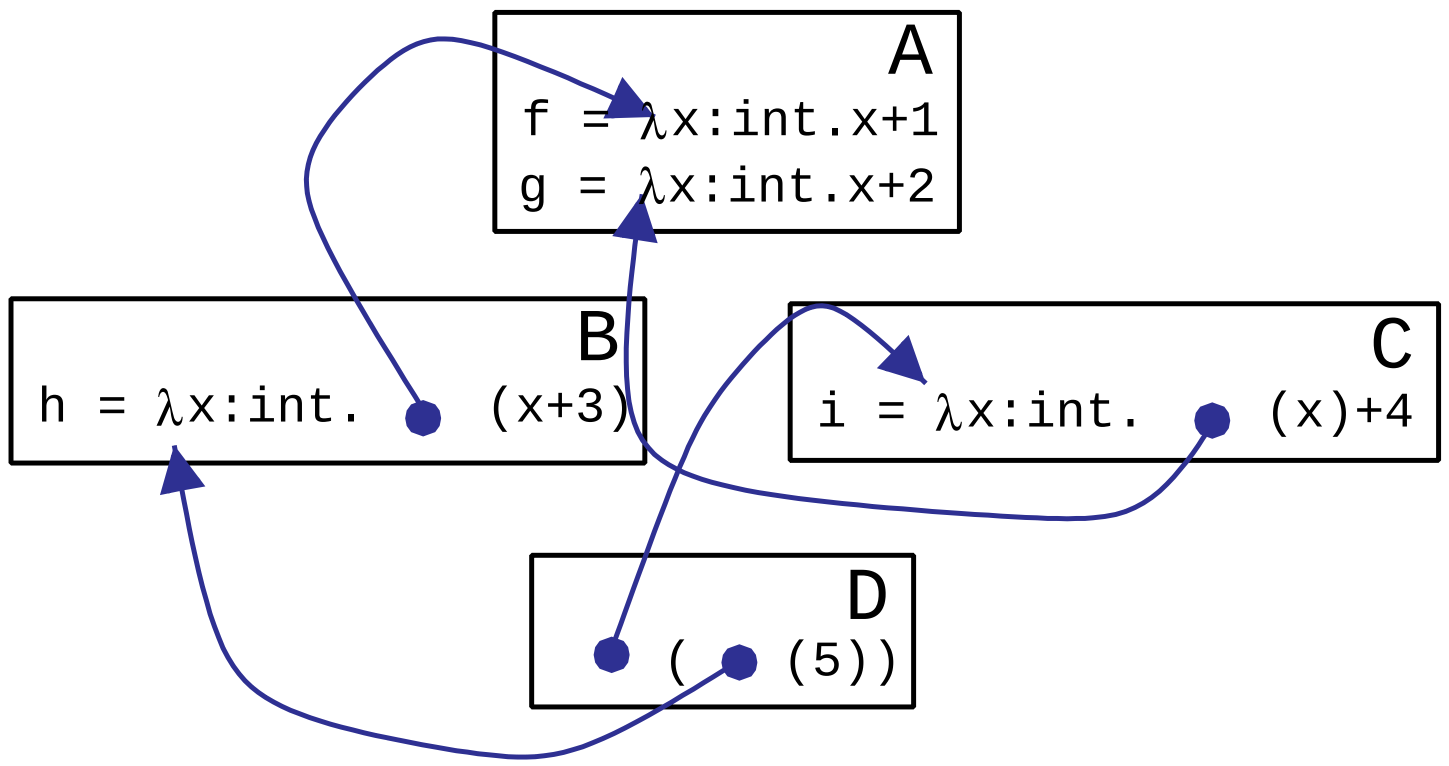
```
B
h = λx:int. A.f(x+3)
```

```
C
i = λx:int. A.g(x)+4
```

```
D
C.i(B.h(5))
```

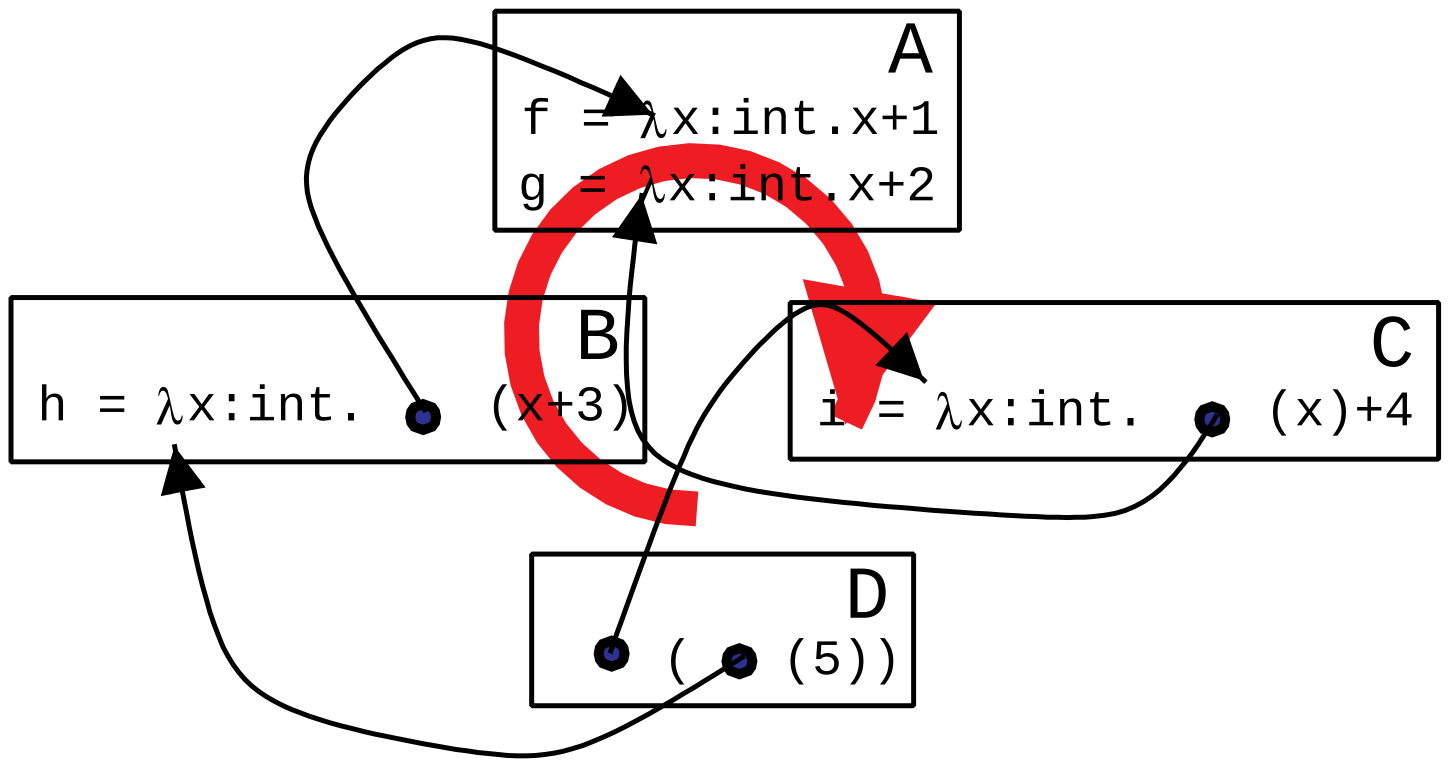
Lifecycle of a program

(1) development (2) linking (3) execution



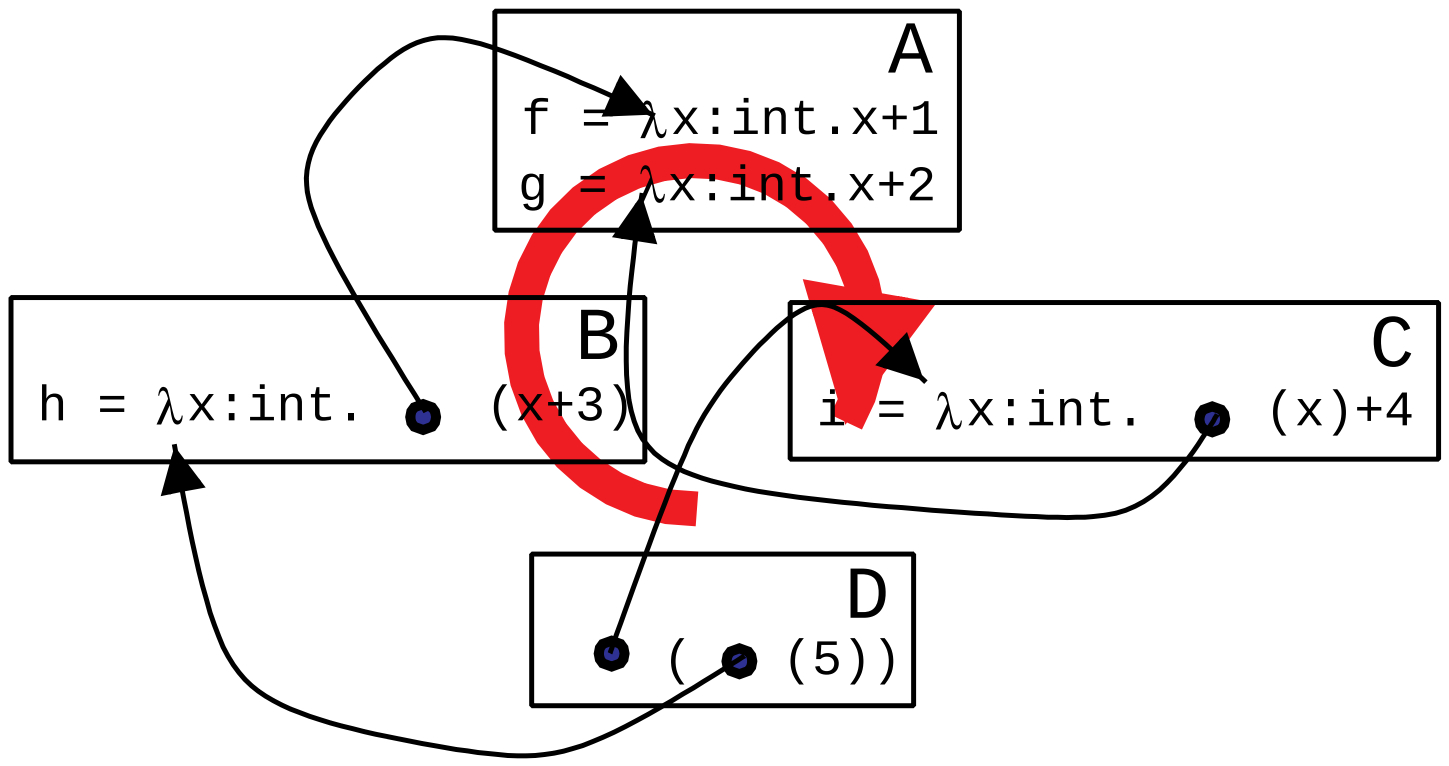
Lifecycle of a program

(1) development (2) linking (3) execution



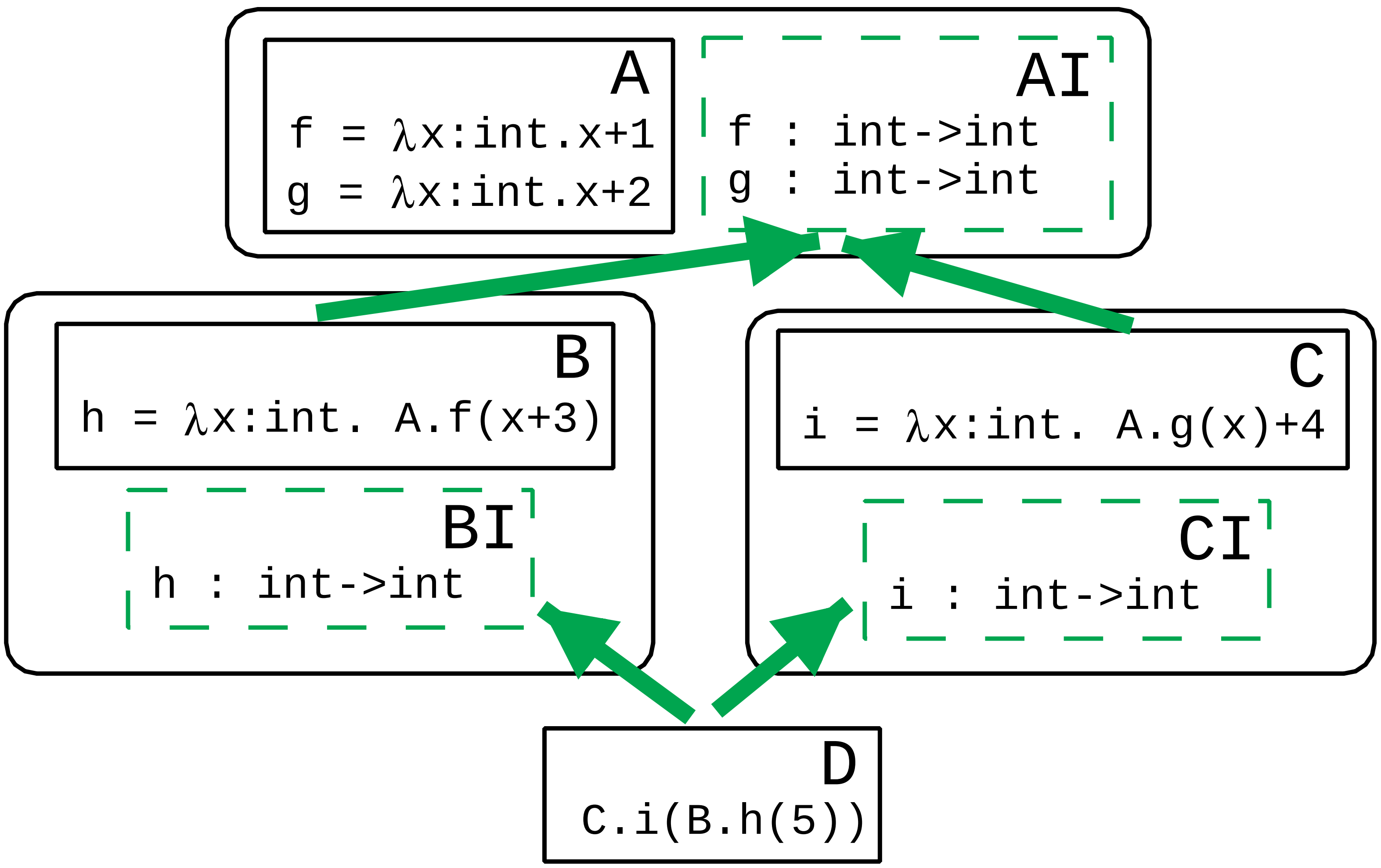
Lifecycle of a program

(1) development (2) linking (3) execution

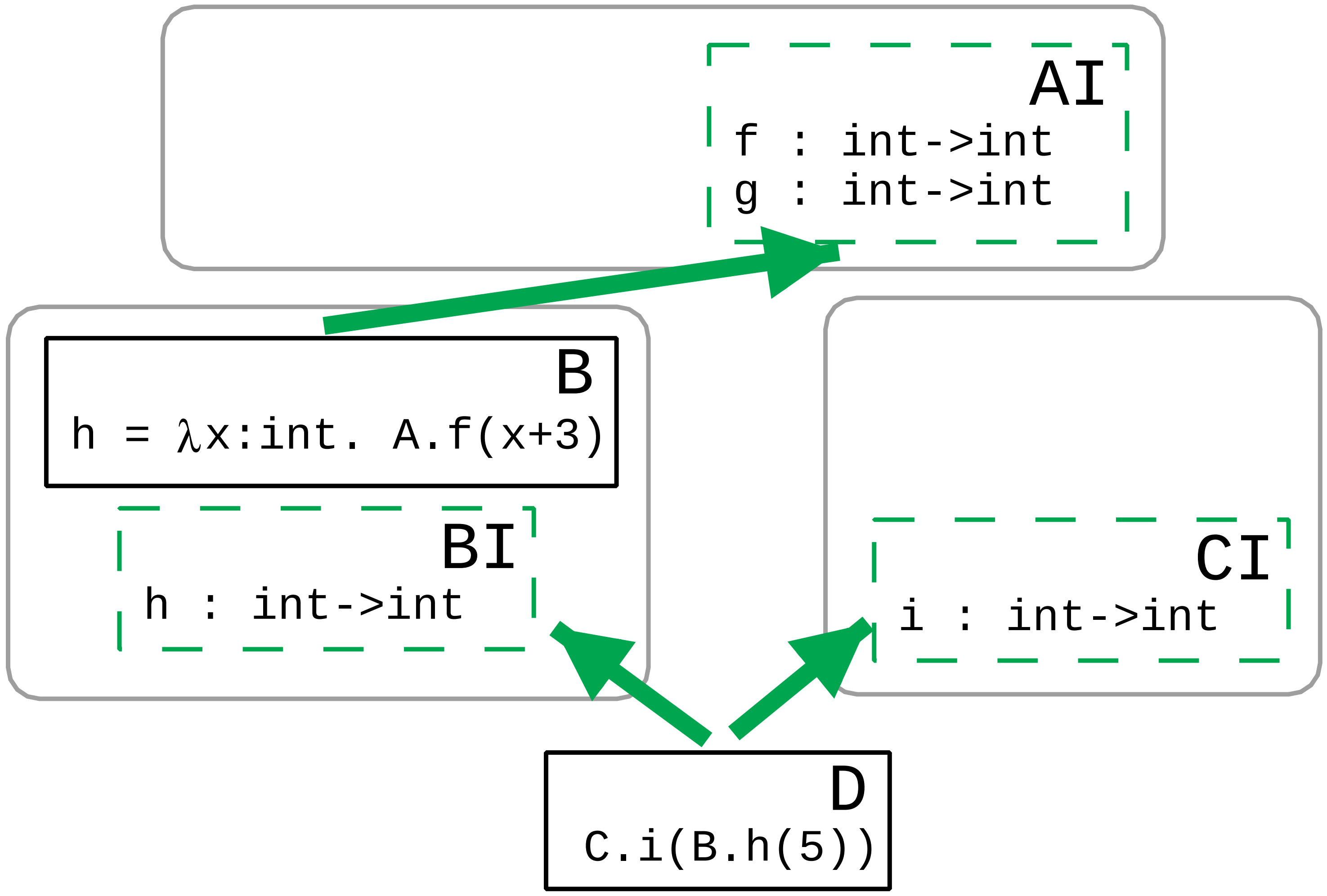


(In practice, the phases are not so neatly separated...)

Interfaces


A
f = λx:int.x+1
g = λx:int.x+2
AI
f : int->int
g : int->int
B
h = λx:int. A.f(x+3)
BI
h : int->int
C
i = λx:int. A.g(x)+4
CI
i : int->int
D
C.i(B.h(5))

Separate development



“True” separate development requires that all dependencies between modules be mediated by explicit interfaces. Modules can then be recompiled in any order.

When modules do depend on each other directly (or, equivalently, when module interfaces are not explicit but are “read off” by the compiler), this dependency induces an ordering on compilation of modules. Changes to modules deep in this ordering will cause “cascading recompilations.”

Hiding Representations

```
A
f = λx:int.x+1
g = λx:int.x+2
```

```
AI
f : int->int
g : int->int
```

Hiding Representations

```
A
f = λx:int.x+1
g = λx:int.x+2
```

```
AI
f : int->int
g : int->int
```

```
A
f = λx:int.x+1
g = λx:int.x+2
```

```
AI
f : int->X
g : X->int
```

Hiding Representations

```
A
f = λx:int.x+1
g = λx:int.x+2
```

```
AI
f : int->int
g : int->int
```

```
A
f = λx:int.x+1
g = λx:int.x+2
```

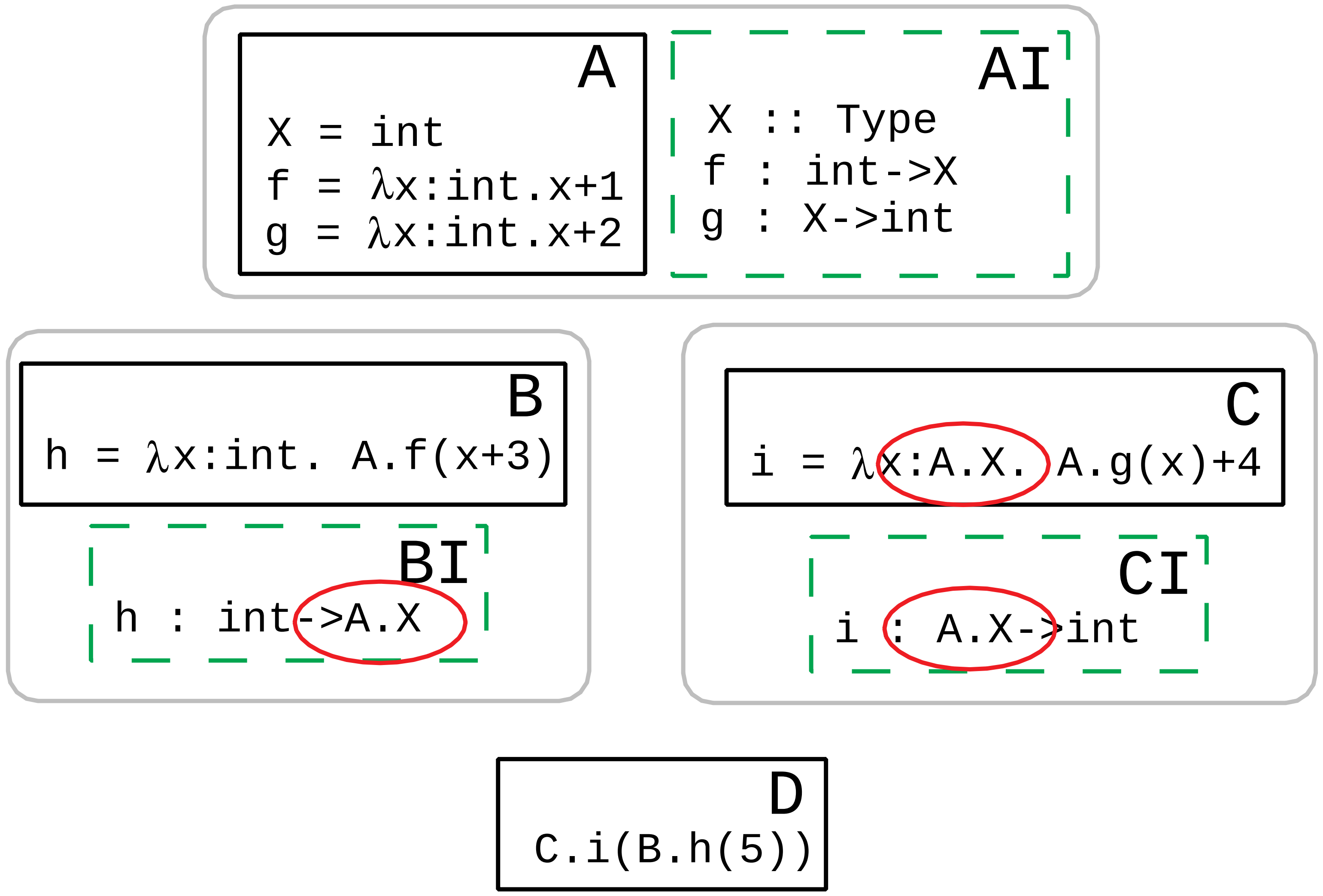
```
AI
f : int->X
g : X->int
```

```
A
X = int
f = λx:int.x+1
g = λx:int.x+2
```

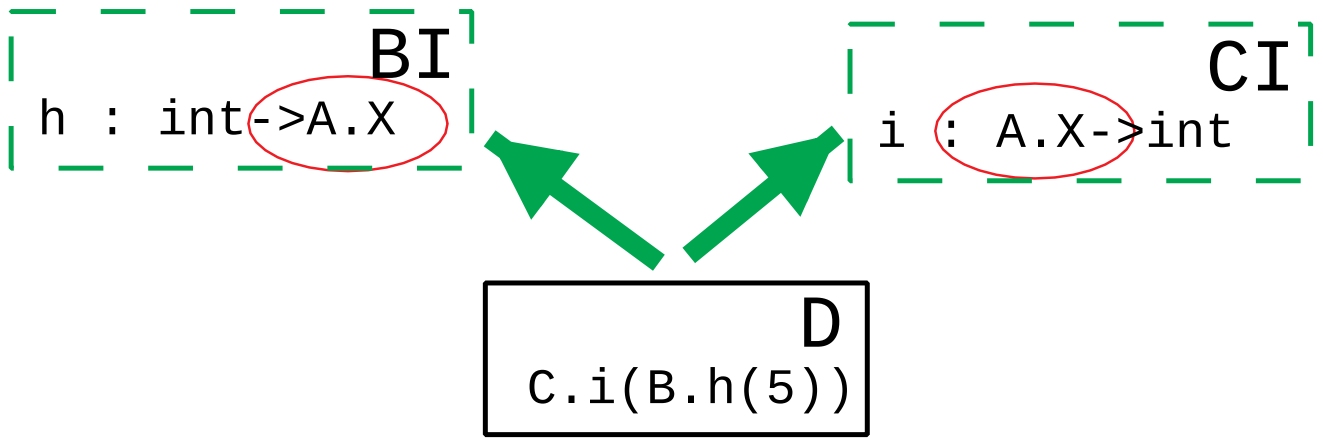
```
AI
X :: Type
f : int->X
g : X->int
```

$\sim$ existential types

Hiding Representations

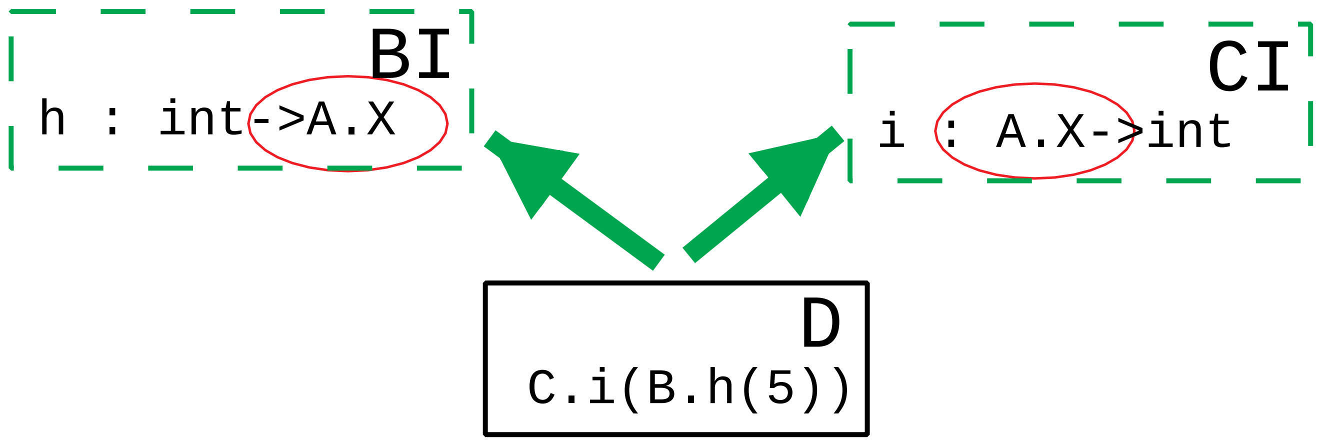

A
```
X = int
f = λx:int.x+1
g = λx:int.x+2
```
AI
```
X :: Type
f : int->X
g : X->int
```
B
```
h = λx:int. A.f(x+3)
```
BI
```
h : int->A.X
```
C
```
i = λx:A.X. A.g(x)+4
```
CI
```
i : A.X->int
```
D
```
C.i(B.h(5))
```

Coherence



When typechecking D, we need to know that the module A mentioned in B's interface and the A mentioned in C's interface are the same (or, at least, that they have the same type component X).

Coherence



When typechecking `D`, we need to know that the module `A` mentioned in `B`'s interface and the `A` mentioned in `C`'s interface are the same (or, at least, that they have the same type component `X`).

Here, this is immediate, since both references to `A` are specific (i.e., they are free vars with the same name)

Generic references

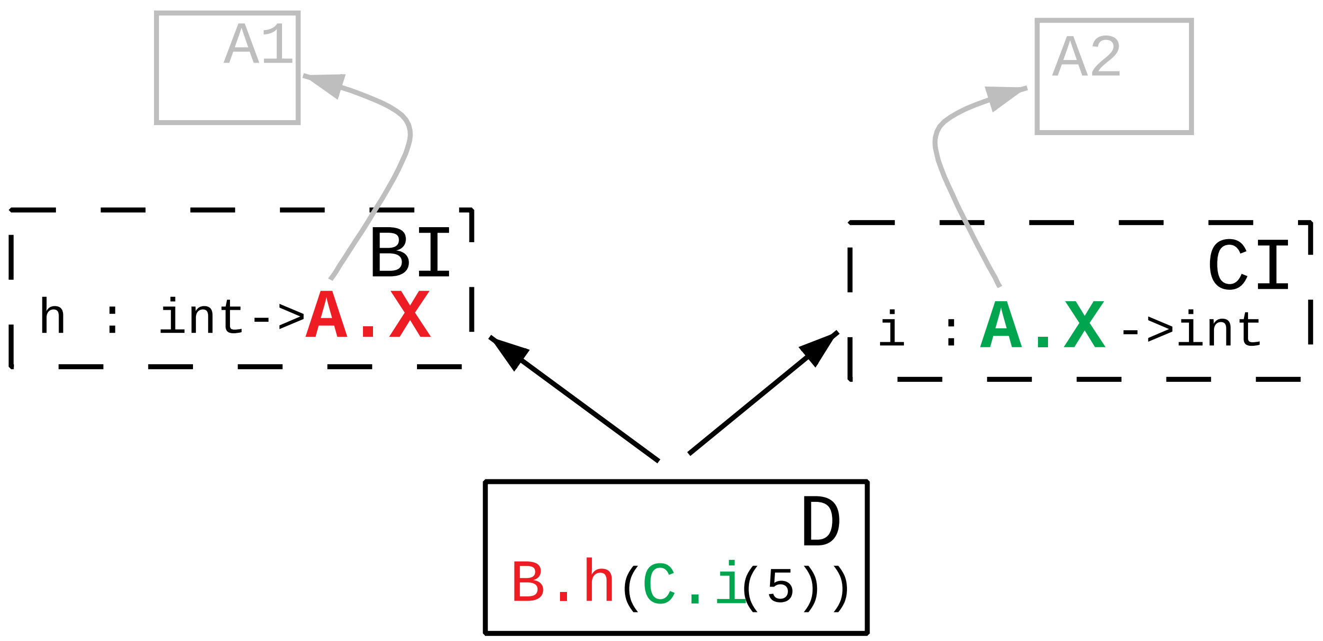
So far, all external references to modules have this specific character. In particular, when an interface refers to another module, it should be interpreted as "the module with this name and interface (whose precise identity will be known at link time)."

This is a key property of simple module systems.

More advanced module systems also support generic references to external modules: we can talk about "a module with such-and-such interface."

However...

Incoherence



If the module name `A` mentioned in `BI` and the `A` mentioned in `CI` might refer to different modules (with different implementations of `X`), then there is no reason why the body of `D` should typecheck!

Forms of Generic References

Various possible realizations of generic references:

- functors (parametric modules)
- multiple class loaders
- etc.

Multiple class loaders

Remember Vijay Saraswat’s tricky Java class-loader bug? [“Java is not Typesafe”, Types posting, 1997]

Essentially, this bug arose from the fact that multiple class loaders in Java give you “a-ness” (generic references to classes) in a language that only understands “the-ness” (specific names for classes).

Furthermore, Sun’s fix [Liang&Bracha] essentially amounted to introducing (dynamically checked) sharing specifications in the run-time system!

Functors

In languages with ML-style modules, generic references come from functors.

These arise in programming in several ways:

1. Fully functorized programming style
2. Multiple implementations of interfaces
3. Generic libraries

However, finding “necessary” examples of functorization is not that easy.

“Fully functorized” style

An early idea in the ML community was that all specific inter-module references should be replaced by generic ones.

Experience showed, though, that the fully functorized style is far too painful to use in practice:

> Making all references generic leads to many spurious coherence issues.

Multiple implementations

Proposal:

Functors arise we want to provide multiple implementations of the same signature: client modules should be parameterized so that we can choose between these implementations at link time.

Example:

The Unison file synchronizer has both a textual and a graphical user interface, both matching the signature `UI`. The main program is parameterized on the user interface module.

Multiple implementations

Proposal:

Functors arise whenever we want to implement the same signature more than once: client modules should be parameterized so that we can link them with alternate implementations.

Not convincing:

Other languages (C, Java, ...) accomplish this by using a direct reference from the main to the UI module and adjusting the “linking context” (search paths, etc.) so that the appropriate implementation is supplied at link time.

Multiple implementations, contd.

Ugh: path hacks!

Much nicer to express linking in a real programming language.

Multiple implementations, contd.

Ugh: path hacks!

Much nicer to express linking in a real programming language.

Counter:

Agreed. But functors are not the only possible “nice linking language.” Why not invent a real language with primitives for manipulating linking contexts in the style of search paths?

(the SML/NJ Compilation Manager goes some distance in this direction...)

Libraries

Proposal:

> Functors can arise when a library module needs to refer to a client module (because the library implementor doesn’t know what name will be chosen for the client module).

Convincing?

Libraries

Proposal:

Functors can arise when a library module needs to refer to a client module (because the library implementor doesn’t know what name will be chosen for the client module).

Convincing?

Somewhat, but perhaps one could also address this sort of application with “path hacks” or a more sophisticated linking language

Multiple simultaneous implementations

Refined proposal:

Functors are needed when we want to implement the same signature multiple times and use more than one of the implementations in the same run of a program.

Multiple simultaneous implementations

Refined proposal:

> Functors are needed when we want to implement the same signature multiple times and use more than one of the implementations in the same run of a program.

Convincing.

E.g.: `Set` module.

Origins of coherence issues

Challenge: Find natural (better yet, common) examples of unavoidable coherence problems.

1. must involve "diamond import" or similar pattern of dependency, with `B` and `C` parameterized on `A`, and `D` on `B` and `C`
2. `A`, `B`, and `C` must have multiple implementations
3. must involve using two (or more) of each implementation in the same run of the program

One example

This functor [from the CMU PsiCo project] computes a delaunay triangulation of a planar structure by projecting it onto a sphere and computing a voronoi diagram there...

```
functor Ruppert
  (structure Geometry2D : GEOMETRY
   structure Geometry3D : GEOMETRY
   sharing type Geometry2D.Number.t = Geometry3D.Number.t
   ...
  )
  = ...
```

1. Diamond import

$$\begin{tikzcd}
 & \texttt{Number} & \\
\texttt{Geometry2D} \arrow[ur] & & \texttt{Geometry3D} \arrow[ul] \\
 & \texttt{Ruppert} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

   The 2-D and 3-D geometries passed to `Ruppert` must share a common representation of numbers used for coordinates.

2. Multiple implementations of GEOMETRY and NUMBER.

3. Multiple simultaneously active instances of GEOMETRY (obviously) and NUMBER (e.g., for multi-precision calculations[Right??]) in a single link-context.

Another example

SML/NJ compiler back end:

1. Code generator is parameterized on machine description. Machine description itself is broken into several parts, depending on common low-level substructures, which must be coherent
2. machine descriptions for many architectures
3. multiple simultaneous machine descriptions present during cross-compilation

Dealing with coherence

To treat such examples (without resorting to dynamic checks), we need deal with the issue of coherence.

Possible approaches:

- dodge the issue by using objects instead of modules
- sharing by parameterization (a.k.a. “sharing by construction,” or “Pebble-style” sharing)
  - parameterization over modules
  - parameterization over types
- sharing by specification, using sharing specifications or where-clauses

Objects

Often, parameterization over modules can be replaced by parameterization over objects (which do not export abstract types, and so do not raise coherence issues).

If

1. a module provides just one abstract type `X`, and
2. the types of all the operations have the form $\mathtt{X} \longrightarrow \mathtt{T}$ or $\mathtt{T} \longrightarrow \mathtt{X}$ (with `X` not in `T`),

then we can re-organize the module as an object.

Example

Recall our module `A`:

```
A = [ X = int                              AI = [ X :: Type
      f = λx:int.x+1                              f : int->X
      g = λx:int.x+2 ]                            g : X->int ]
```

Here are `A` and `AI` in Java:

```
class A implements AI {
    int rep;                                interface AI {
    A(int x) { rep = x+1; }                    A (int x);
    int g () { return rep+4; }}                int g ();  }
```

Representation Hiding, O-O style

Roughly:

- Operations taking the hidden type as parameter become methods with one less parameter: the parameter of hidden type is represented by the “implicit parameter” `self` or `this`.
- Operations returning the hidden type become constructors of the class.

Limitation: No simultaneous abstractions

Example:

```
AbsynI = [ Ty :: Type,
           Tm :: Type,

           lam : Ty -> (Tm -> Tm) -> Tm,
           app : Tm -> Tm -> Tm,
           ... ]
```

Translating this into OO style is awkward.

Limitation: Binary methods

Binary operations do not fit well with the object-oriented programming style.

- Each object carries its own representation (and associated operations)
- “Deep” binary operations—ones that require privileged access to the concrete representations of two different abstract values—do not fit this model.

(N.b.: the PsiCo and SML/NJ examples from a few slides ago were not “objectifiable” for this reason.)

Digression: Binary Methods and Classes

But… O-O languages like Java do support binary methods, don't they?

Digression: Binary Methods and Classes

But... O-O languages like Java do support binary methods, don’t they?

Yes. In classes, not interfaces.

Relies on the fact that all the instances of a class (and its subclasses) carry the same fields. So if a parameter belongs to the same class as the current object, then it must have (at least) the same fields.

What Java does not support is generic references to classes.

Sharing by parameterization

Idea: Parameterize signatures wrt. all references to external modules (i.e., change external references from free to λ-bound names)

Sharing by parameterization

```
A  = [ X = int,    f = ...,     g = ...    ]
AI = [ X :: Type,  f : int->X,  g : X->int ]

B  =  λA:AI.     [ h = λx:int. A.f(x+3) ]
BI =  [A:AI] -> [ h : int -> A.X ]

C  =  λA:AI.     [ i = λx:A.X. A.g(x)+4 ]
CI =  [A:AI] -> [ h : A.X -> int ]

D  =  λA:AI. λB:BI(A). λC:CI(A).
          [ C.i(B.h(5)) ]
```

Sharing by specification

Idea: Augment signatures with substructures corresponding to all external names

Sharing by specification

```
A  = [ X = int,     f = ...,      g = ...    ]
AI = [ X :: Type,   f : int->X,   g : X->int ]

B  = [  A:AI = A,   h = λx:int. A.f(x+3) ]
BI = [  A : AI,     h : int -> A.X ]

C  = [  A:AI = A,   i = λx:A.X. A.g(x)+4 ]
CI = [  A : AI,     i : A.X -> int ]

D  =  λB:BI. λC:CI with A=B.A.
        [ C.i(B.h(5)) ]
```

Sharing by specification

Abstract substructures in signatures function as placeholders for names of specific modules in future sharing specifications.

Comparisons

Key point: The augmented signatures appearing in the sharing-by-specification style are signatures! (A parameterized signature is not a signature: it is a function from modules to signatures.)

“Post-hoc parameterization” as needed

Using sharing-by-parameterization, the only way to be “parametric enough” is to parameterize all external references in case they need to be shared later.

Difficulties

Two forms of sharing by parameterization:

- Parameterization over external modules (as above)

  Observation [MacQueen]: As dependency hierarchies become deeper, interfaces parameterized on modules scale badly.

- Parameterization just over the abstract types from external modules (e.g., Haskell)

  Works.

Parameterization over modules: Flat version

```
M1 : I1 = [T :: Type, f : ...]

M2 : I2 = [T :: Type, f : ...M1.T...]

M3 : I3 = [T :: Type, f : ...M2.T...]

M4 : I4 = [T :: Type, f : ...M3.T...]

etc.
```

Parameterized version

Signatures must be "maximally parameterized," anticipating arbitrary patterns of sharing.

```
I1 = [T :: Type, f : ...]

I2 = λM1:I1. [T :: Type, f : ...M1.T...]

I3 = λM1:I1. λM2:I2(M1).
       [T :: Type, f : ...M2.T...]

I4 = λM1:I1. λM2:I2(M1). λM3:I3(M1)(M2).
       [T :: Type, f : ...M3.T...]
```

Parameterized version

Signatures must be “maximally parameterized,” anticipating arbitrary patterns of sharing.

```
I1 = [T :: Type, f : ...]

I2 = λM1:I1. [T :: Type, f : ...M1.T...]

I3 = λM1:I1. λM2:I2(M1).
        [T :: Type, f : ...M2.T...]

I4 = λM1:I1. λM2:I2(M1). λM3:I3(M1)(M2).
        [T :: Type, f : ...M3.T...]
```

Header information grows as square of dependency depth.

The final nail in the coffin

Now suppose we decide to change `M1` so that its interface depends on a new module `M0`...

```
I0 = [T :: Type, f : ...]
I1 = λM0:I0. [T :: Type, f : ...M0.T...]
I2 = λM0:I0. λM1:I1(M0).
       [T :: Type, f : ...M1.T...]
I3 = λM0:I0. λM1:I1(M0). λM3:I2(M0)(M1).
       [T :: Type, f : ...M2.T...]
I4 =  λM0:I0. λM1:I1(M0).
         λM2:I2(M0)(M1). λM3:I3(M0)(M1)(M2).
       [T :: Type, f : ...M3.T...]
```

Requires $(\text{dependency depth})^2$ changes to existing code!

Parameterization over types

Instead of abstracting signatures on modules, we can abstract just on the types from these modules (which are all we really need to refer to anyway):

```
I1 = [T1 :: Type, f : ...]

I2 = λT1::Type. [T2 :: Type, f : ...T1...]

I3 = λT2::Type. [T :: Type, f : ...T2...]

I4 = λT3::Type. [T :: Type, f : ...T3...]
```

Parameterization over types

Much better: no quadratic growth of header information!

But: still suffers from “anticipatory parameterization”

Effectively, this style amounts to performing “phase separation” manually.

Bottom line

Coherence requirements are fundamentally equations between types (or structures)

Two ways to handle them:

- Deal with them directly in the type theory of the language (sharing by specification).

  Elaborator deals with this rich type theory by compiling it down to something simpler.

- Make the language simpler by shifting the work of elaboration to the programmer (sharing by parameterization on types).

Conclusions

- Generic inter-module references are a key design choice.
- Industrial languages get along without them by using search path hacks, objects, etc., but lose expressiveness that matters in some real examples.
- Allowing them raises the issue of coherence.
- Coherence can be dealt with either “automatically” (sharing by specification) or “manually” (sharing by parameterization on types).
- Parameterization on modules does not scale.