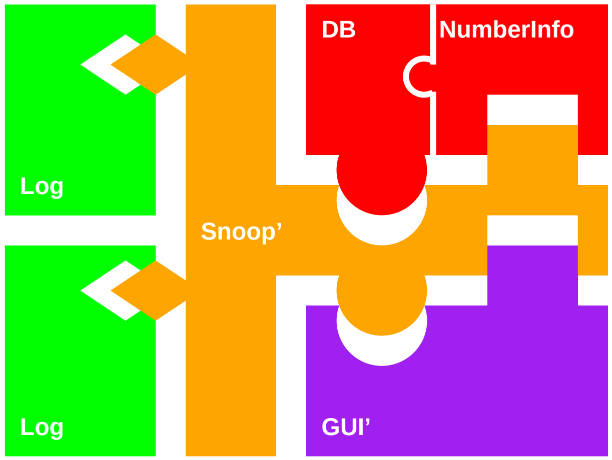
Programming Language Support for Software Components

Matthew Flatt

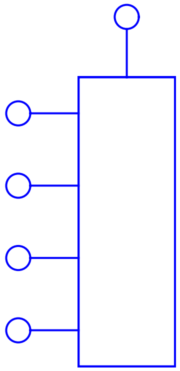
University of Utah

Software Components



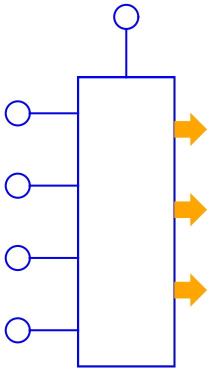
COM Objects as Components

Exports defined *statically*



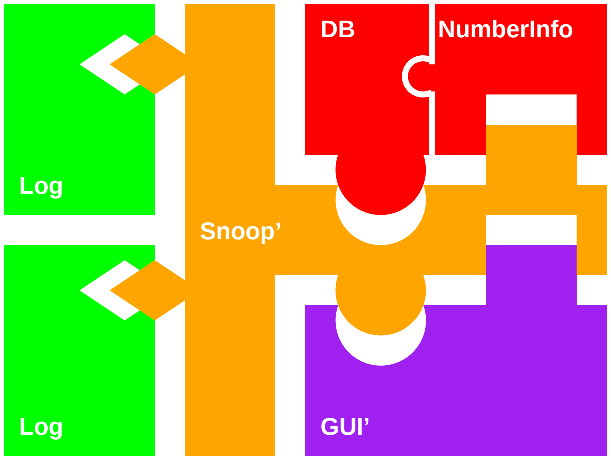
COM Objects as Components

Exports defined *statically*



Imports defined *dynamically*
(through arbitrarily complex, uncheckable C code)

Software Components

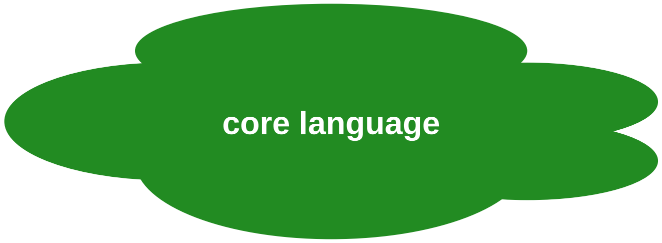


Component Properties

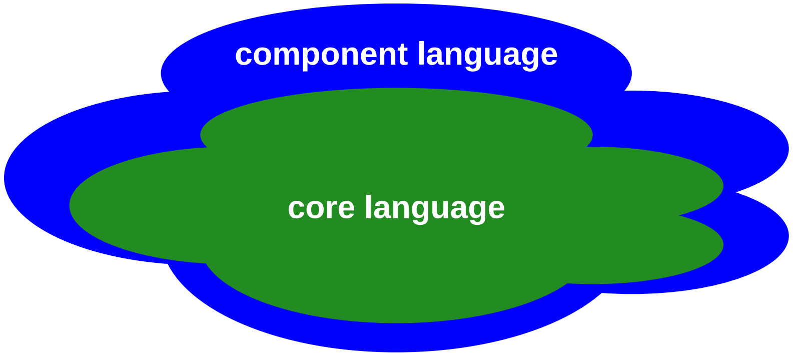
- Each component has a well-defined interface
- Each component can be separately checked and compiled
- Interface specifies the *shape* of imports, not the *source*
- Components can be instantiated multiple times
- Component linking is hierarchical
- Components can have mutual dependencies (recursion)
- Linking specifications are static and checked

➡ Language support for components

Component Languages



Component Languages



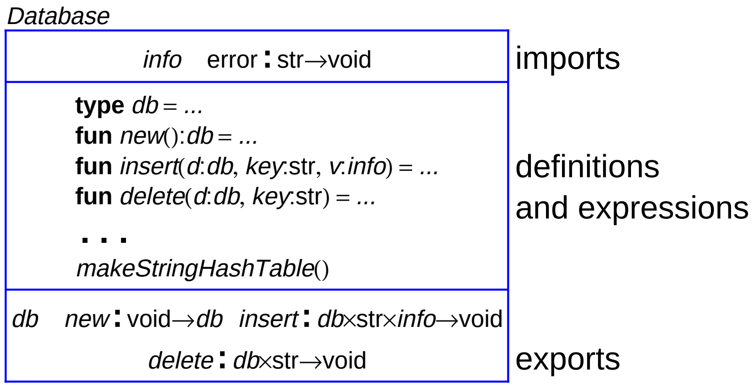
Implemented Component Languages

- **DrScheme** : a component extension of Scheme
  - Robert Bruce Findler, Shriram Krishnamurthi, Matthias Felleisen, John Clements, Paul Steckler, Cormac Flanagan (then @Rice)
  - `http://www.drscheme.org/`
- **Knit** : a component language for C
  - Alastair Reid, Eric Eide, Jay Lepreau, Leigh Stoller (@Utah)
  - `http://www.cs.utah.edu/flux/alchemy/knit/`
- **Jiazzi** : a component language for Java
  - Sean McDirmid, Wilson Hsieh (@Utah)
  - `http://www.cs.utah.edu/plt/jiazzi/`

Outline

- **Software Components**
- ➤ **Unit Model of Software Components**
- **Components and Classes**
- **Jiazzi: Components in Java**
- **Components for Systems Software**
- **Related Work, Open Problems, Conclusion**

Unit Definitions



Database

info error : str→void

imports

type *db* = ...
fun *new*():*db* = ...
fun *insert*(*d*:*db*, *key*:str, *v*:*info*) = ...
fun *delete*(*d*:*db*, *key*:str) = ...
. . .
makeStringHashTable()

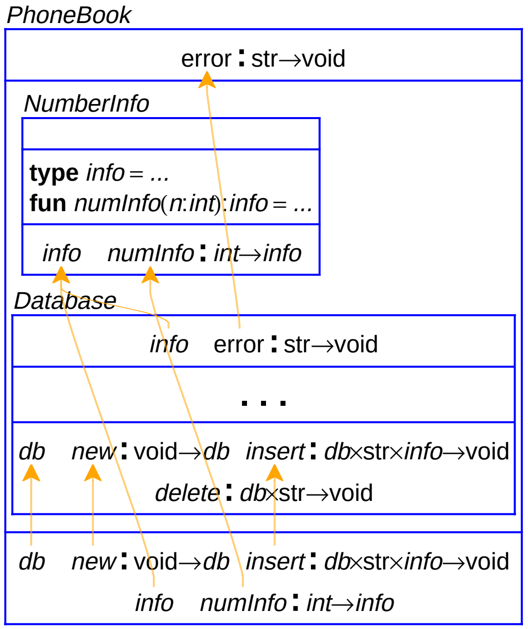
definitions and expressions

db *new* : void→*db* *insert* : *db*×str×*info*→void
delete : *db*×str→void

exports

- Imported and exported variables have types
- Type expressions for variables can use imported and exported types

Linking Units



PhoneBook

error : str→void

NumberInfo

type *info* = ...
fun *numInfo*(*n*:*int*):*info* = ...

info *numInfo* : *int*→*info*

Database

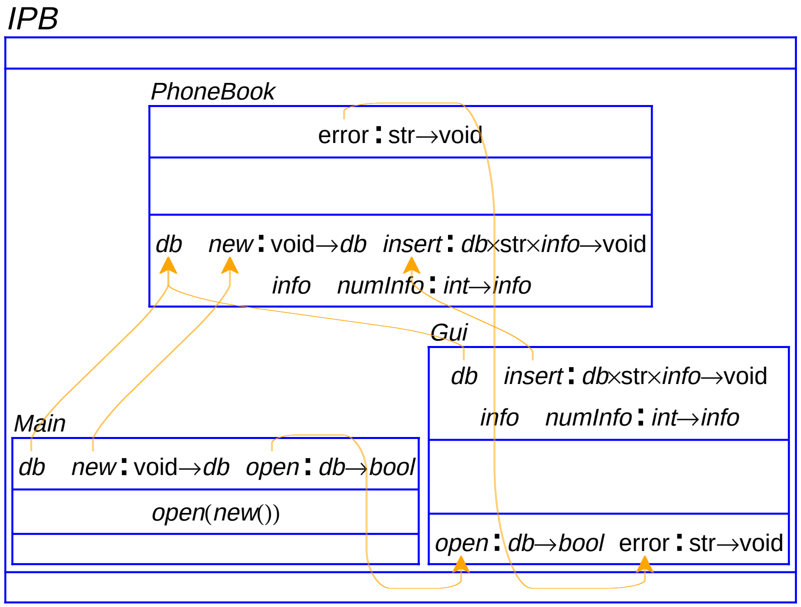
info error : str→void

. . .

db *new* : void→*db* *insert* : *db*×str×*info*→void
delete : *db*×str→void

db *new* : void→*db* *insert* : *db*×str×*info*→void
info *numInfo* : *int*→*info*

A Complete Program



IPB

PhoneBook

error **:** str→void

db *new* **:** void→*db* *insert* **:** *db*×str×*info*→void

info *numInfo* **:** *int*→*info*

Gui

db *insert* **:** *db*×str×*info*→void

info *numInfo* **:** *int*→*info*

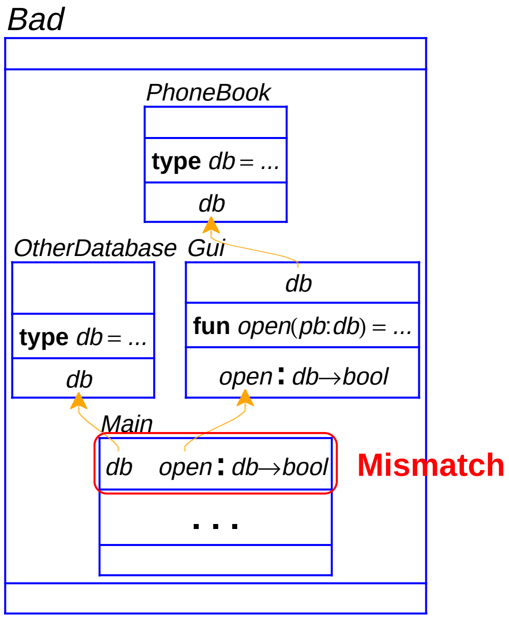
open **:** *db*→*bool* error **:** str→void

Main

db *new* **:** void→*db* *open* **:** *db*→*bool*

open(*new*())

An Ill-formed Linkage



Bad

PhoneBook

type *db* = ...

db

OtherDatabase

type *db* = ...

db

Gui

db

fun *open*(*pb*:*db*) = ...

open **:** *db*→*bool*

Main

db *open* **:** *db*→*bool*

. . .

Mismatch

Unit Summary

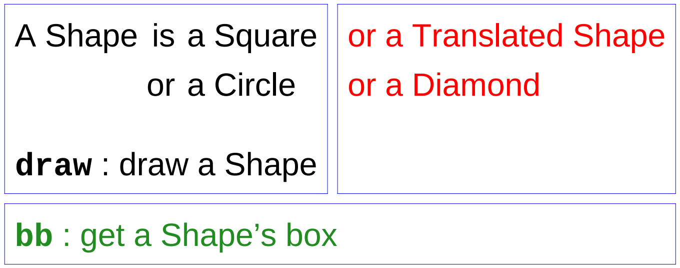
- Well-defined import and export interfaces
- Explicit linking, external to the linked unit
- Hierarchical linking through compound units
- Static checking of links

Full model also covers dynamic linking

Outline

- **Software Components**
- **Unit Model of Software Components**
- ➤ **Components and Classes**
- **Jiazzi: Components in Java**
- **Components for Systems Software**
- **Related Work, Open Problems, Conclusion**

Expressiveness of Components and Classes



A Shape is a Square
or a Circle

or a Translated Shape
or a Diamond

draw : draw a Shape

bb : get a Shape's box

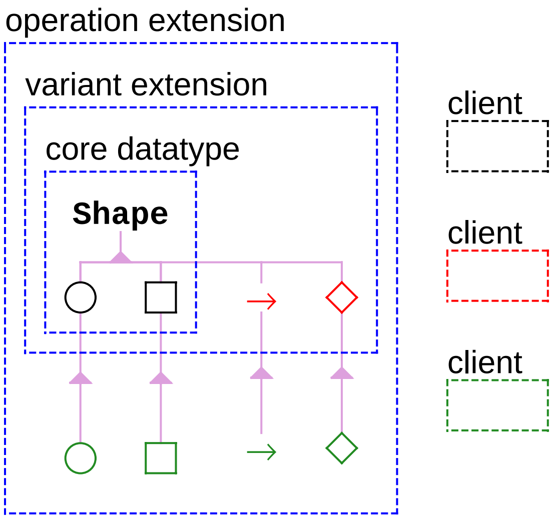
Without modifying

○ core implementation
○ clients

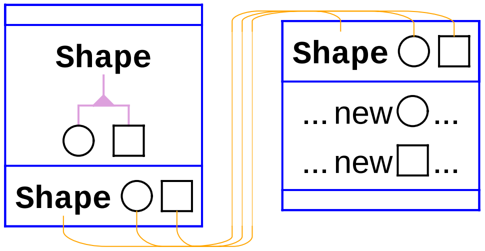
Other Work on Extensibility

- Steele, 1994
- Felleisen and Cartwright, 1994
- Liang, Hudak, and Jones, 1994
- Duggan and Sourelis, 1996
- Palsberg and Jay, 1997
- Kuhne, 1997
- Krishnamurthi, Felleisen, and Friedman, 1998
- Clifton, Leavens, Chambers, and Millstein, 2000
- Zenger and Odersky, 2001

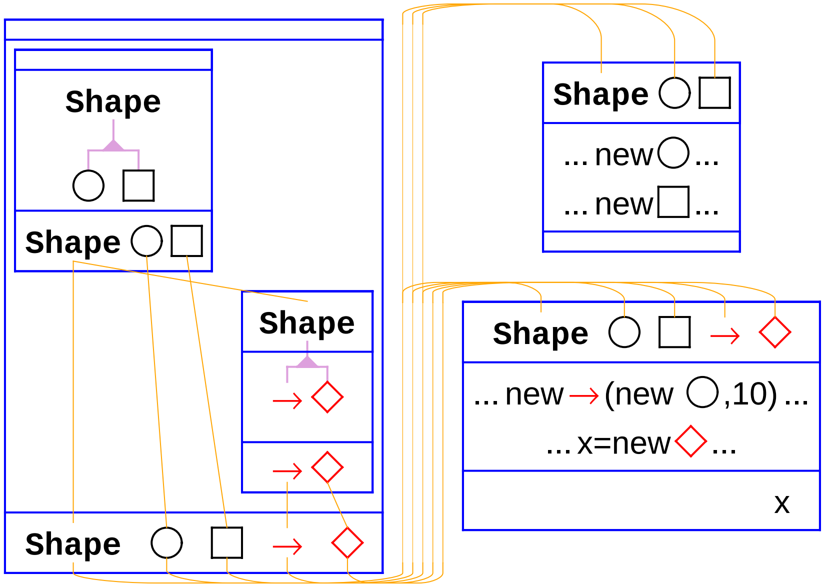
Componential Extension



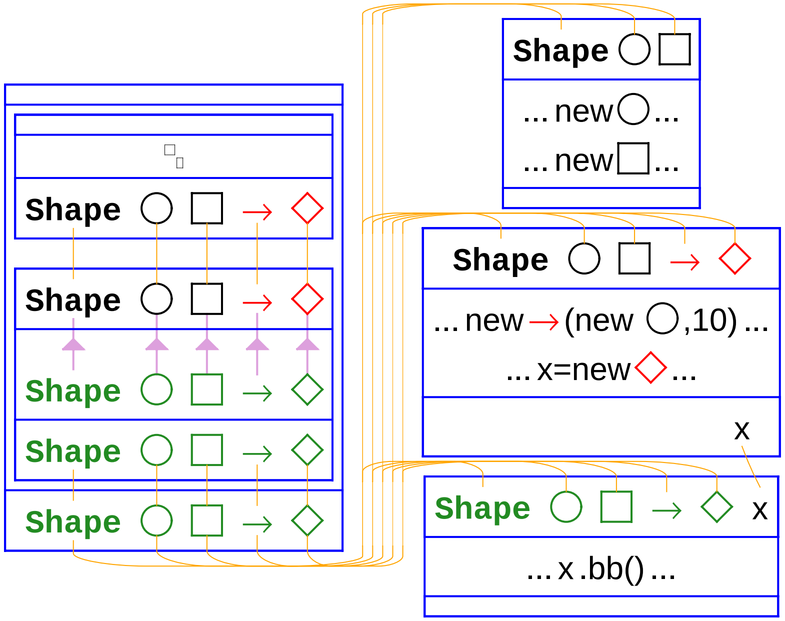
Original Datatype and Client



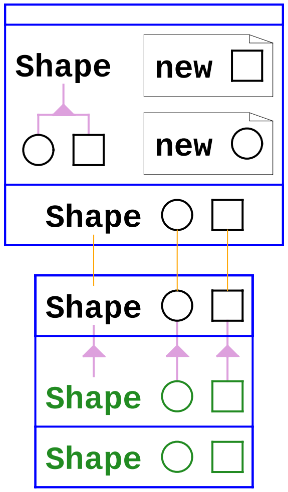
Variant Extension



Operation Extension

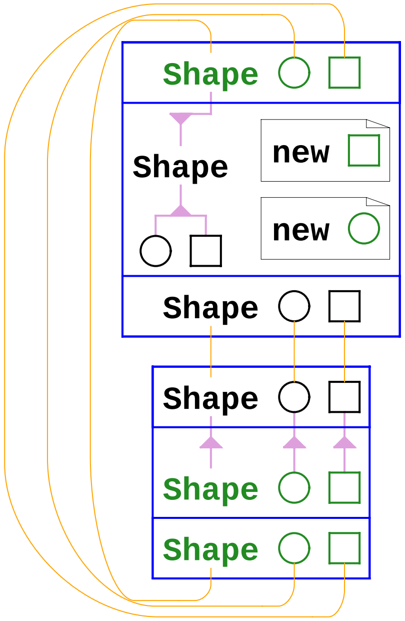


Self-Instantiation of the Datatype



instantiates
wrong classes

Self-Instantiation of the Datatype

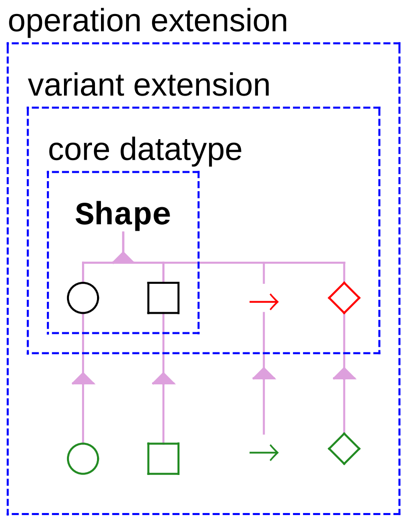


fix with the
open class
pattern

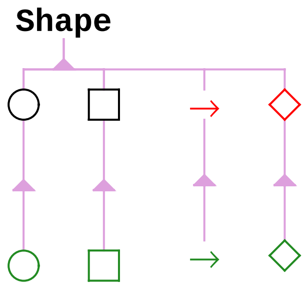
Extensibility through Classes and Units

- Allows both variant and operation extension
- No modification (or recompilation) of existing modules
- No programmer-maintained indirections
- Natural: resulting structure matches a monolithic solution

Solution's Natural Structure



Solution's Natural Structure



Outline

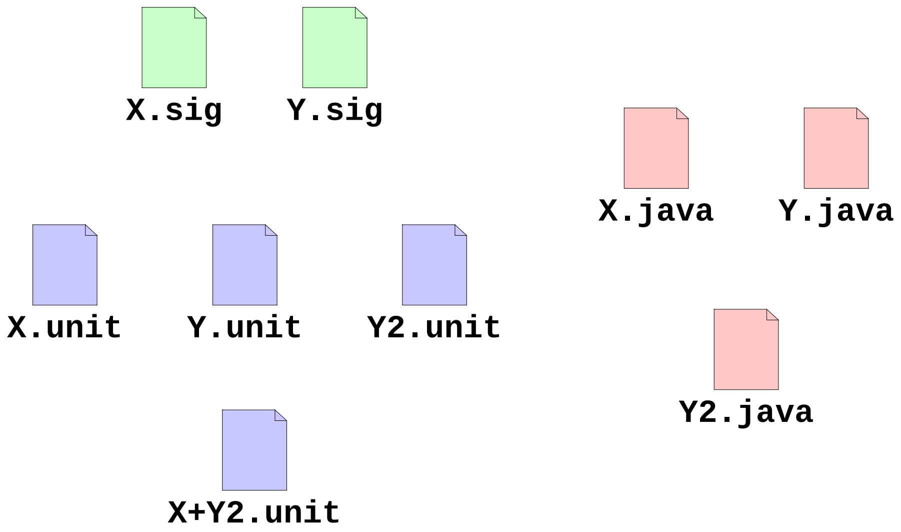
- Software Components
- Unit Model of Software Components
- Components and Classes
- ➤ Jiazzi: Components in Java
- Components for Systems Software
- Related Work, Open Problems, Conclusion

Jiazzi: Components for Java

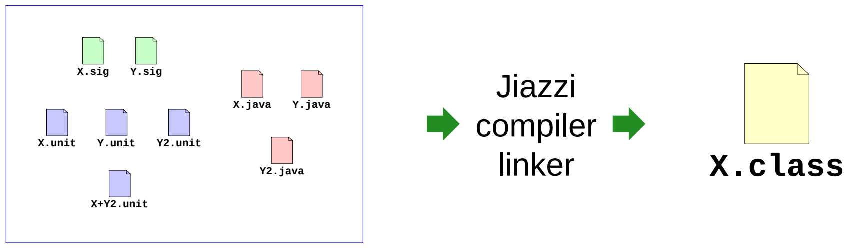
Issues for a realistic, statically typed language:

- Integrating with existing infrastructure
- Defining component signatures
- Avoiding method collisions

Programming with Jiazzi


X.sig
Y.sig
X.java
Y.java
X.unit
Y.unit
Y2.unit
Y2.java
X+Y2.unit

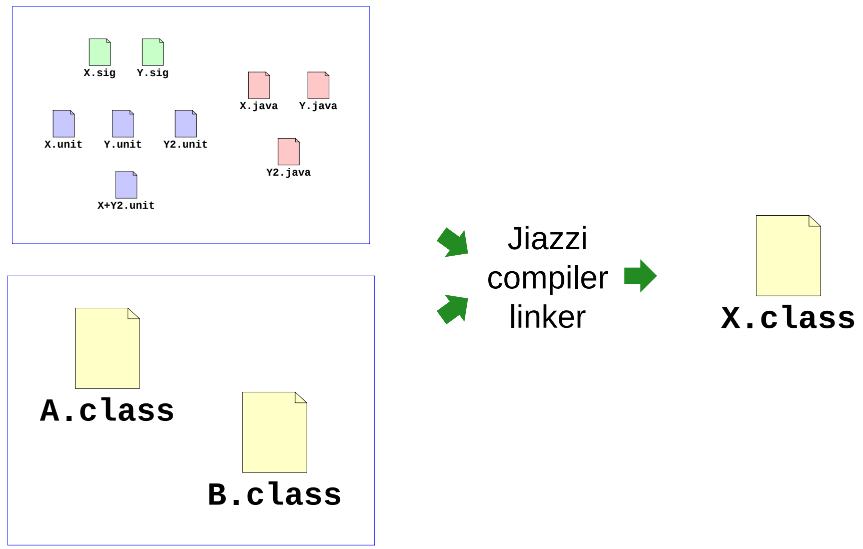
Programming with Jiazzi


X.sig
Y.sig
X.java
Y.java
X.unit
Y.unit
Y2.unit
Y2.java
X+Y2.unit

Jiazzi compiler linker

X.class

Programming with Jiazzi


X.sig
Y.sig
X.java
Y.java
X.unit
Y.unit
Y2.unit
Y2.java
X+Y2.unit

A.class
B.class

Jiazzi compiler linker

X.class

Jiazzi Signature Syntax

Almost:

```
signature shapes_s {
  class Shape ≤ Object { ... }
  class ○ ≤ Shape { ... }
  class □ ≤ Shape { ... }
}
```

- Where does **Object** come from?
- What if we need to instantiate ○ and □ ?
- What if **Shape** needs to be extended before ○ and □ ?

Jiazzi Signature Syntax

Correct:

```
signature shapes_s<lang_p, fixpt_p> {
  class Shape ≤ lang_p.Object { ... }
  class ○ ≤ fixpt_p.Shape { ... }
  class □ ≤ fixpt_p.Shape { ... }
}
```

Jiazzi Signature Syntax

Signature of the variant extension:

```
signature more_shapes_s<lang_p, shapes_p> {
  class → ≤ shapes_p.Shape { ... }
  class ◇ ≤ shapes_p.Shape { ... }
}
```

Jiazzi Signature Syntax

Signature of the operation extension:

```
signature bbox_shapes_s<lang_p, shapes_p> {
  class Shape ≤ shapes_p.Shape { ... }
  class ○ ≤ shapes_p.○ { ... }
  class □ ≤ shapes_p.□ { ... }
  class → ≤ shapes_p.→ { ... }
  class ◇ ≤ shapes_p.◇ { ... }
}
```

Jiazzi Signature Syntax

Signature of an operation extension for **shrink**:

```
signature shrink_shapes_s<lang_p, shapes_p, fixpt_p> {
  class Shape ≤ shapes_p.Shape { ... }
  class ○ ≤ shapes_p.○ {
    ... fixpt_p.○ shrink( int scale); ...
  }
  class □ ≤ shapes_p.□ { ... }
  class → ≤ shapes_p.→ { ... }
  class ◇ ≤ shapes_p.◇ { ... }
}
```

Jiazzi Unit Syntax

Less extensible version (can’t extend **Shape** early):

```
atom Shapes {
  export shapes_out : shape_s<[java.lang], shapes_out>;
}
/* sources "Shape.java", "Circle.java", "Square.java" */
```

Jiazzi Unit Syntax

More extensible version:

```
atom Shapes {
  import shapes_in : shape_s<[java.lang], shapes_in>;
  export shapes_out : shape_s<[java.lang], shapes_in>;
}
/* sources "Shape.java", "Circle.java", "Square.java" */

atom Draw {
  import shapes_in : shape_s<[java.lang], shapes_in>;
  export draw_out : draw_s<[java.lang], shapes_in>;
}
/* sources "Draw.java" */
```

Jiazzi: Components for Java

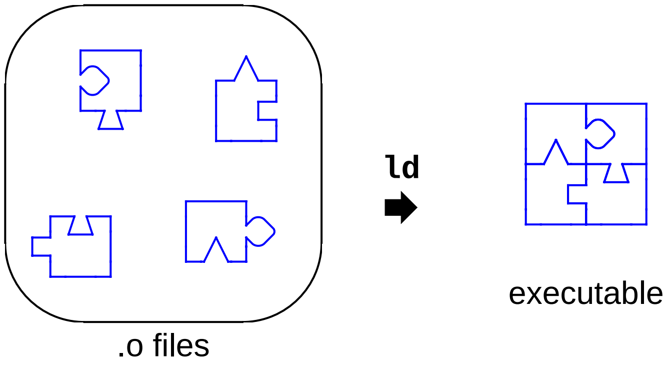
Issues for a realistic, statically typed language:

- Integrating with existing infrastructure
- Defining component signatures
- Avoiding method collisions

Outline

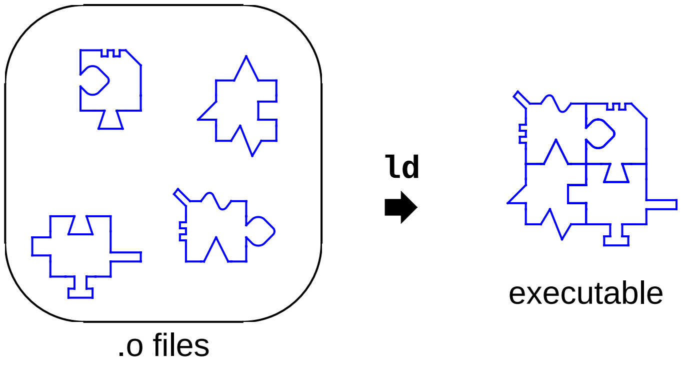
- Software Components
- Unit Model of Software Components
- Components and Classes
- Jiazzi: Components in Java
- ➤ Components for Systems Software
- Related Work, Open Problems, Conclusion

Components for Systems Software



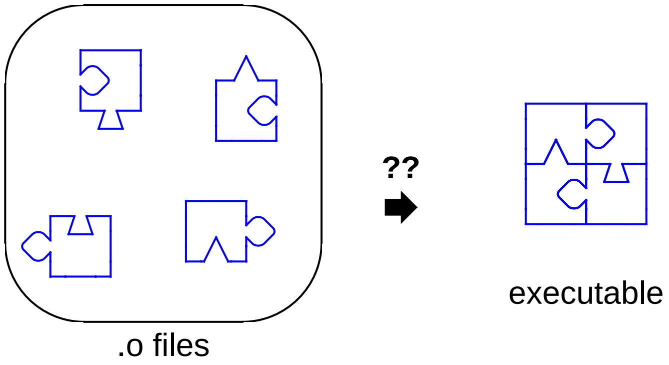
- Most low-level software is implemented in C
- Compiled object (.o) files act as components

Components for Systems Software



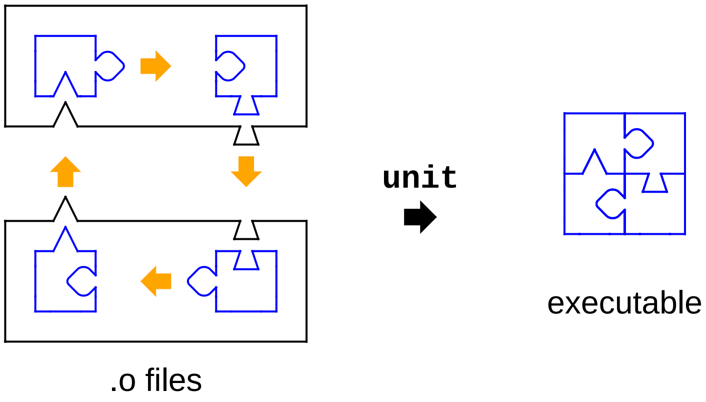
- The boundaries and requirements of .o files are typically not obvious

Components for Systems Software



- Some linking patterns cannot be expressed

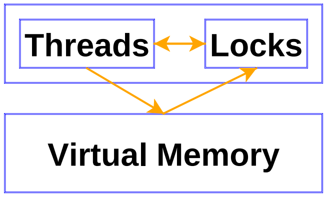
Components for Systems Software



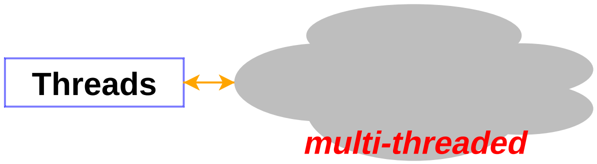
- Units add boundary specifications, replace the linker

Low-Level Compositions

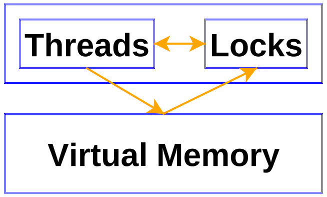
- Initialization order is fragile



- Performance is crucial
- Non-local properties need to be checked



Initialization



- Programmer supplies local dependencies:

```
      import₁ ... importₙ
definition₁                  init-dep₁
...                           ...
definitionₘ                  init-depₚ
      export₁ ... exportₖ
```

- Linker schedules globally

Performance

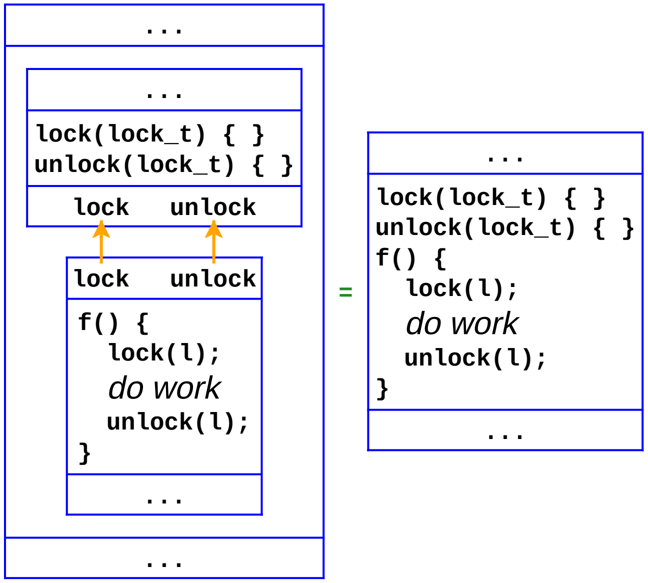
Performance goal:

- To make aggressive componentization practical
- Not to speed up existing code

Achieve by optimizing across component boundaries

  - Relies on static nature of linking

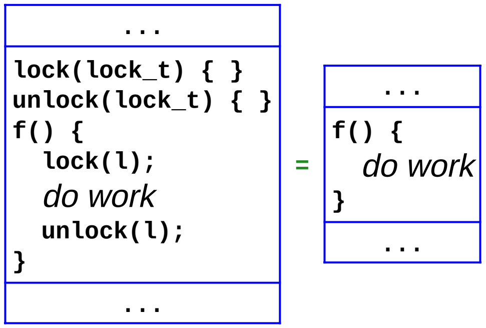
Performance



```
...
...
lock(lock_t) { }
unlock(lock_t) { }
lock    unlock
lock    unlock
f() {
  lock(l);
  do work
  unlock(l);
}
...
...
```

```
...
lock(lock_t) { }
unlock(lock_t) { }
f() {
  lock(l);
  do work
  unlock(l);
}
...
```

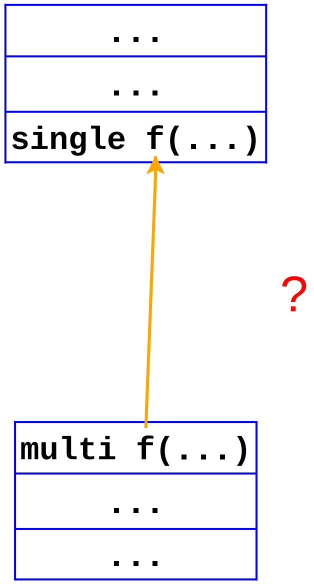
Performance



```
...
lock(lock_t) { }
unlock(lock_t) { }
f() {
  lock(l);
  do work
  unlock(l);
}
...
```

```
...
f() {
  do work
}
...
```

Non-Local Checking

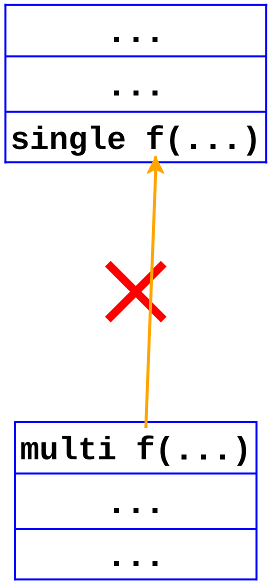


```
...
...
single f(...)
```

```
multi f(...)
...
...
```

- Type-like anotations can detect mismatches

Non-Local Checking

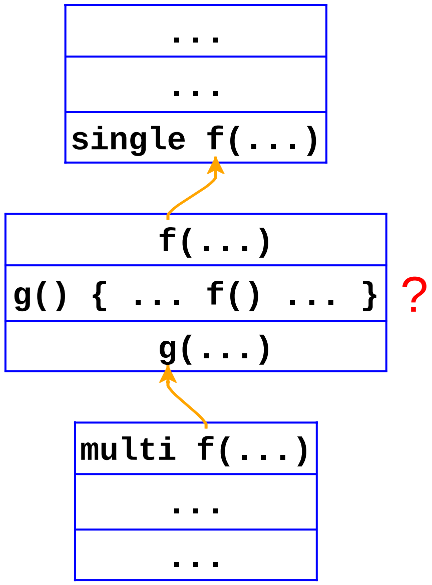


```
...
...
single f(...)
```

```
multi f(...)
...
...
```

- Type-like anotations can detect mismatches

Non-Local Checking


...

...

single f(...)

f(...)

g() { ... f() ... } ?

g(...)

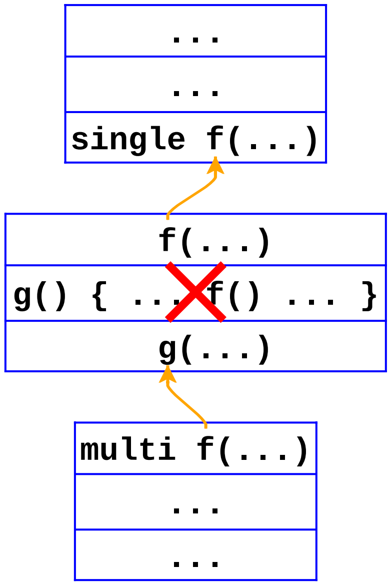
multi f(...)

...

...

- Also need to detect indirect mismatches

Non-Local Checking


...

...

single f(...)

f(...)

g() { ... f() ... }

g(...)

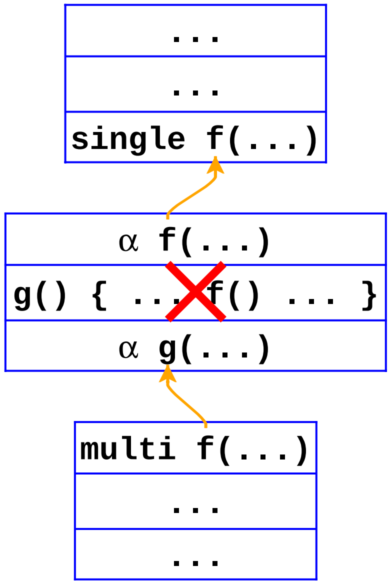
multi f(...)

...

...

- Also need to detect indirect mismatches

Non-Local Checking


...

...

single f(...)

α f(...)

g() { ... f() ... }

α g(...)

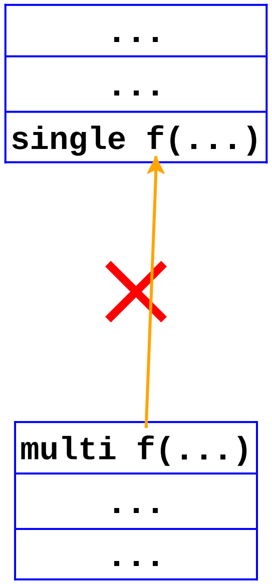
multi f(...)

...

...

- Also need to detect indirect mismatches

Non-Local Checking


...

...

single f(...)

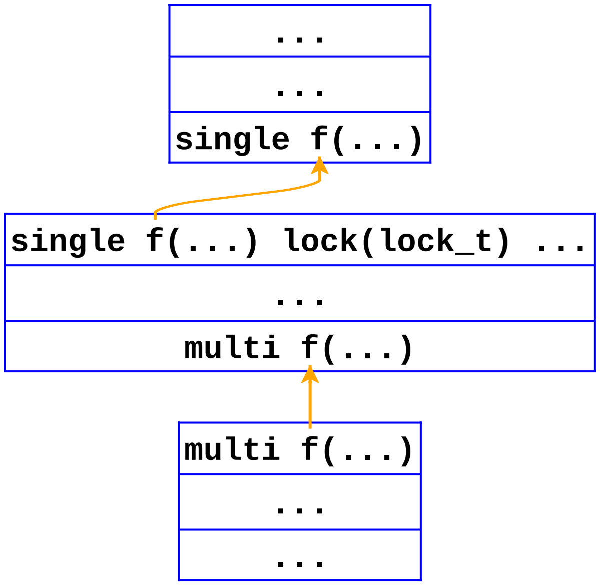
multi f(...)

...

...

- Can also automate mismatch repairs

Non-Local Checking



- Can also automate mismatch repairs

Low-Level Components

- Systems code benefits from an explicit component language
- Additional practical concerns for low-level code require extensions to the basic unit model

Knit and the OSKit

- OSKit version "1": used .o files for all components
  - couldn’t create certain combinations
- OSKit version "2": used COM for many components
  - too dynamic; errors reported late
  - significant overhead
- OSKit version "3": uses units for most components
  - initial results are promising
  - still refining the language

Outline

- **Software Components**
- **Unit Model of Software Components**
- **Components and Classes**
- **Jiazzi: Components in Java**
- **Components for Systems Software**
- ➤ **Related Work, Open Problems, Conclusion**

Related Work

- McIlroy
- Szyperski: *Component Software*
- Cedar/Mesa (Xerox PARC)
- MacQueen, Harper, Crary, *et al.*: ML modules
- Ancona and Zucca
- Bracha

Open Problems

- Interoperability among core languages
- Effective specification of non-type properties
- Resource control sensitive to component boundaries

Conclusion

> *Beyond object-oriented programming*
> - Szyperski

- Units: a programming language for components
  - expressive
  - checkable
  - practical: DrScheme, Knit, Jiazzi, ...