Backpack: Retrofitting Haskell with Interfaces

Scott Kilpatrick
MPI-SWS

Derek Dreyer
Simon Peyton Jones
Simon Marlow

MPI-SWS
Microsoft Research
Facebook

POPL ’14
22 January 2014
Strong Modularity

Typecheck B *separately* from A by depending on an interface A.
Strong Modularity

Typecheck B *separately* from A by depending on an interface A.
Strong Modularity

Typecheck B separately from A by depending on an interface A.

Reuse B by instantiating it with different implementations of A.
Strong Modularity

Typecheck B \textit{separately} from A by depending on an interface A.

Reuse B by instantiating it with different implementations of A.
Strong Modularity

Typecheck B separately from A by depending on an interface A.

Reuse B by instantiating it with different implementations of A.

Typecheck mutually dependent modules before tying the recursive knot.
Weak Modularity

(implementations depend on implementations)

• at Haskell’s module level

Socket

Server
Weak Modularity
(implementations depend on implementations)

- at Haskell’s module level
- at Haskell’s package level
Weak Modularity
(implementations depend on implementations)

• at Haskell’s module level

![Diagram showing Socket connected to Server]

• at Haskell’s package level

![Diagram showing P-1.1, 1.2, 1.3, 1.4, and Q-2]

no interfaces ⇒ no strong modularity
Haskell has weak modularity.

We want to extend it with strong modularity.
Haskell has weak modularity.

We want to extend it with *strong* modularity.

Solution: Retrofit
Haskell has weak modularity.

We want to extend it with strong modularity.

**Solution:**

Design strong layer on top of weak modules!

Retrofit
“Why not just use the ML module system?” (i.e. functors)
“Why not just use the ML module system?” (i.e. functors)

- a lot of work on the ML module system
- interfaces via signatures, reuse via functors
“Why not just use the ML module system?” (i.e. functors)

• a lot of work on the ML module system
• interfaces via signatures, reuse via functors

but
“Why not just use the ML module system?” (i.e. functors)

**good point**
- a lot of work on the ML module system
- interfaces via signatures, reuse via functors

**but**
- functors won’t work for us
  - incompatible with recursive linking
  - unclear how to incorporate into weak modules
“Why not just use the ML module system?” (i.e. functors)

**good point**
- a lot of work on the ML module system
- interfaces via signatures, reuse via functors

**but**
- functors won’t work for us
  - incompatible with recursive linking
  - unclear how to incorporate into weak modules
- functors are ill-suited for separate compilation
  - ML langs require additional system on top
Introducing Backpack

• Retrofits Haskell with strong modularity
  - Designed at package level
  - Employs simplified mixin design
  - Defined as elaboration into weak Haskell modules
  - Separate typechecking, not separate compilation
Outline

- Language tour
- Semantics
- Future work
modules in today’s Haskell
Modules in Today’s Haskell

Socket.hs

module Socket where
  data SocketT = ...
  open = ...

Server.hs

module Server where
  import Socket
  data ServerT = ...SocketT...
Modules in Today’s Haskell

Socket.hs

module Socket where
    data SocketT = ...
    open = ...

Server.hs

module Server where
    import Socket
    data ServerT = ...SocketT...
Modules in Today’s Haskell

Socket.hs

module Socket where
  data SocketT = ...
  open = ...

Server.hs

module Server where
  import Socket
  data ServerT = ...SocketT...
Modules in Today’s Haskell

Socket.hs

module Socket where
  data SocketT = ... 
  open = ...

Server.hs

module Server where
  import Socket
  data ServerT = ...SocketT...
Modules in Today’s Haskell

Socket.hs

module Socket where
  data SocketT = ...
  open = ...

Server.hs

module Server where
  import Socket
  data ServerT = ...SocketT...
Modules in Today’s Haskell

Socket.hs

module Socket where
  data SocketT = ...
  open = ...

Server.hs

module Server where
  import Socket
  data ServerT = ...SocketT...
modules in today’s Haskell
Boot Files: Almost Interfaces

Socket.hs-boot

module Socket where
  data SocketT
  open :: Int -> SocketT

- implementation mechanism in GHC compiler
- used as “forward declaration” for recursive modules
Boot Files: Almost Interfaces

Socket.hs-boot

module Socket where

data SocketT

open :: Int -> SocketT

• implementation mechanism in GHC compiler
• used as “forward declaration” for recursive modules
Boot Files: Almost Interfaces

Socket.hs-boot

module Socket where
  data SocketT
  open :: Int -> SocketT

• implementation mechanism in GHC compiler
• used as “forward declaration” for recursive modules
modules in today’s Haskell
Packages in Backpack

Socket.hs

module Socket where
  data SocketT = ...
  open = ...

Server.hs

module Server where
  import Socket
  data ServerT = ...SocketT...
Packages in Backpack

```haskell
package complete-server where
  Socket = [data SocketT = ...]
  open = ...
  Server = [import Socket
             data ServerT = ...SocketT...]
```

Identity Variables

\(\alpha, \beta \in \text{IdentVars}\)

Identity Constructors

\(K \in \text{IdentCtors}\)

Identities

\(\nu ::= \alpha | K \nu | \mu \alpha \nu\)

Identity Substitutions

\(\phi, \theta ::= \{\alpha : \nu\}\)
Packages in Backpack

package complete-server where

Socket = [data SocketT = ...]
open = ...

Server = [import Socket
data ServerT = ...SocketT...]

package server where

Socket = [Socket.hs]
Server = [Server.hs]

package serverimpl where

include socketsig

package server-linked where

include serverimpl
include socketimpl

package server-linked-alt where

include serverimpl
include socketimpl-alt

Identity Variables

α, β ∈ IdentVars

Identity Constructors

K ∈ IdentCtors

Identities

ν ::= α | Kν | µα.ν

Identity Substitutions

φ, θ ::= {α : ν}
Packages in Backpack

```haskell
package complete-server where

Socket = [data SocketT = ...]
open = ...

Server = [import Socket
data ServerT = ...SocketT...]
```

Identity Variables

\( \alpha, \beta \in \text{IdentVars} \)

Identity Constructors

\( K \in \text{IdentCtors} \)

Identities

\( \nu ::= \alpha | K \nu | \mu \alpha. \nu \)

Identity Substitutions

\( \phi, \theta ::= \{ \alpha ::= \nu \} \)
Packages in Backpack

```haskell
package complete-server where
  Socket = [data SocketT = ...]
    open = ...
  Server = [import Socket
    data ServerT = ...SocketT...]
```
Packages in Backpack

```
package complete-server where
Socket = [data SocketT = ...]
open = ...
Server = [import Socket
data ServerT = ...SocketT...]
```

Identity Variables

\( \alpha, \beta \in \text{IdentVars} \)

Identity Constructors

\( K \in \text{IdentCtors} \)

Identities

\( \nu ::= \alpha | K \nu | \mu \alpha.\nu \)

Identity Substitutions

\( \phi, \theta ::= \{ \alpha : = \nu \} \)

Packages in Backpack
(With Holes)

package partial-server where
Socket :: [data SocketT
    open :: Int -> SocketT]
Server = [import Socket
    data ServerT = ...SocketT...]

package complete-server where
Socket = [data SocketT = ...
    open = ...]
Server = [import Socket
    data ServerT = ...SocketT...]

package server where
Socket = [Socket.hs]
Server = [Server.hs]

package serverimpl where
include socketsig
Server = [...

package server-linked where
include serverimpl
include socketimpl

package server-linked-alt where
include serverimpl
include socketimpl-alt

Identity Variables
\( \alpha, \beta \in \text{IdentVars} \)

Identity Constructors
\( K \in \text{IdentCtors} \)

Identities
\( \nu ::= \alpha | K \nu | \mu \alpha. \nu \)

Identity Substitutions
\( \phi, \theta ::= \{ \alpha ::= \nu \} \)

Package Names
\( P \in \text{PkgNames} \)

Module Path Names
\( p \in \text{ModPaths} \)

Package Repositories
\( R ::= D_1, ..., D_n \)

Package Definitions
\( D ::= \text{package}\ \ Pt \ \ where \ B_1, ..., B_n \)

Bindings
\( B ::= p = [M] | p :: [S] | p = p | \text{include}\ Pt r \)

Thinning Specs
\( t ::= (p_1, ..., p_n) \)

Renaming Specs
\( r ::= \{ p_1 \rightarrow p_1', ..., p_n \rightarrow p_n' \} \)

Module Expressions
\( M ::= \text{imports}; \text{exports}; \text{defns} \)

Signature Expressions
\( S ::= \text{imports}; \text{decls} \)
Packages in Backpack
(With Holes)

package partial-server where

Socket :: [data SocketT
open :: Int -> SocketT]

Server = [import Socket
data ServerT = ...SocketT...]

package complete-server where

Socket = [data SocketT = ...
open = ...]

Server = [import Socket
data ServerT = ...SocketT...]

package server where

Socket = [Socket.hs]
Server = [Server.hs]

package serverimpl where

include socketsig

Server = [...

package server-linked where

include serverimpl
include socketimpl

package server-linked-alt where

include serverimpl
include socketimpl-alt

K Ser KSock Kalt A K B α µ α.K A(K B(α))
µ α.K B(K A(α))
µ α.K B(K A(α))
Packages in Backpack
(With Holes)

package partial-server where

Socket ::  [data SocketT
open :: Int -> SocketT ]

Server =  [import Socket
data ServerT = ...SocketT...]

Socket

Server
Packages in Backpack
(With Holes)

- **package** partial-server **where**
  - `Socket` :: [data `SocketT` open :: Int -> `SocketT`]
  - `Server` = [import `Socket` data `ServerT` = ...`SocketT`...]

- **package** complete-server **where**
  - `Socket` :: [data `SocketT` open = ...]
  - `Server` = [import `Socket` data `ServerT` = ...`SocketT`...]

- **package** server **where**
  - `Socket` = [Socket.hs]
  - `Server` = [Server.hs]

- **package** serverimpl **where**
  - include `socketsig` Server = [...

- **package** server-linked **where**
  - include `serverimpl` include `socketimpl`
Package Inclusion

package complete-server where
Socket = [data SocketT = ...]
    [open = ...]
Server = [import Socket
    [data ServerT = ...SocketT...]]
package socketimpl where
  Socket = [data SocketT = ... ]
          [open = ... ]

package complete-server where
  include socketimpl
  Server = [import Socket]
           [data ServerT = ... SocketT... ]
package socketimpl where

Socket = [data SocketT = ...]
[open = ...]

package complete-server where

include socketimpl

Server = [import Socket
[data ServerT = ...SocketT...]]
Package Inclusion

package socketimpl where
Socket = [data SocketT = ...]
open = ...

package complete-server where
include socketimpl
Server = [import Socket
  data ServerT = ...SocketT...]

15
**Package Inclusion**

```
package socketimpl where
    Socket = [data SocketT = ... , open = ... ]

package complete-server where
    include socketimpl
    Server = [import Socket
               data ServerT = ...SocketT... ]
```
Package Inclusion

package socketimpl where

Socket = [data SocketT = ...]
open = ...

package complete-server where
include socketimpl

Server = [import Socket
data ServerT = ...SocketT...]

Socket

Server

package
package
Package Inclusion

package socketimpl where
Socket = [data SocketT = ...]
open = ...

package complete-server where
include socketimpl
Server = [import Socket]
data ServerT = ...SocketT...

Package Inclusion

package socketsig where
  Socket :: [data SocketT
  open :: Int -> SocketT ]

package partial-server where
  include socketsig
  Server = [import Socket
            data ServerT = ...SocketT...]

3

4
package socketsig where
Socket :: [data SocketT
            [open :: Int -> SocketT]]

package partial-server where
    include socketsig
Server = [import Socket
            [data ServerT = ...SocketT...]]
package socketsig where
Socket :: [data SocketT
open :: Int -> SocketT ]

package partial-server where
(include socketsig
Server = [import Socket
data ServerT = ...SocketT...
]
package socketsig where
Socket :: [data SocketT
          open :: Int -> SocketT ]

package partial-server where
include socketsig
Server = [import Socket
           data ServerT = ...SocketT...]

Package Inclusion
**Package Inclusion**

```haskell
package socketsig where
    Socket :: [data SocketT
               open :: Int -> SocketT ]

package partial-server where
    include socketsig
    Server = [import Socket
               data ServerT = ...SocketT...]
```

![Diagram](image)
Linking

package socketsig where
  Socket :: [data SocketT
             open :: Int -> SocketT ]

package partial-server where
  include socketsig
  Server = [import Socket
             data ServerT = ...SocketT... ]

package socketimpl where
  Socket = [data SocketT = ...]
            [open = \n             ... ]

package main where
  include partial-server
  include socketimpl
Linking

package socketsig where
Socket :: [data SocketT
open :: Int -> SocketT ]

package partial-server where
include socketsig
Server = [import Socket
data ServerT = ...SocketT... ]

package socketimpl where
Socket = [data SocketT = ... ]
open = \n ... ]

package main where
include partial-server
include socketimpl
Linking

package socketsig where
  Socket :: [data SocketT
            open :: Int -> SocketT ]

package partial-server where
  include socketsig
  Server = [import Socket
             data ServerT = ...SocketT... ]

package socketimpl where
  Socket = [data SocketT = ...]
             [open = λn. ... ]

package main where
  include partial-server
  include socketimpl

Socket

Socket

Server

Socket

Server
Linking

package socketsig where
  Socket :: [data SocketT
             open :: Int -> SocketT]

package partial-server where
  include socketsig
  Server = [import Socket
             data ServerT = ...SocketT...]

package socketimpl where
  Socket = [data SocketT = ...]
             [open = \n               ...]

package main where
  include partial-server
  include socketimpl
Linking

package socketsig where
    Socket :: [data SocketT
    open :: Int -> SocketT ]

package partial-server where
    include socketsig
    Server = [import Socket
    data ServerT = ...SocketT...
    open = ...

package socketimpl where
    Socket = [data SocketT = ...
    open = \n    Server
    data ServerT = ...SocketT...

package main where
    include partial-server
    include socketimpl
Linking

package socketsig where
Socket :: [data SocketT
open :: Int -> SocketT ]

package partial-server where
include socketsig
Server = [import Socket
data ServerT = ...SocketT... ]

package socketimpl where
Socket = [data SocketT = ...]
open = λn. ...

package main where
include partial-server
include socketimpl

[data SocketT = ...] ⊑ [data SocketT
open = λn. ... ] ⊑ [data SocketT
open :: Int -> SocketT ]
package socketsig where
Socket :: [data SocketT
open :: Int -> SocketT ]

package partial-server where
include socketsig
Server = [import Socket
data ServerT = ...SocketT... ]

package socketimpl where
Socket = [data SocketT = ...
open = λn. ... ]

package main where
include partial-server
include socketimpl
Reuse

```haskell
package server-linked-1 where
  include partial-server
  include socketimpl-1

package server-linked-2 where
  include partial-server
  include socketimpl-2
```
Reuse

```plaintext
package server-linked-1 where
  include partial-server
  include socketimpl-1

package server-linked-2 where
  include partial-server
  include socketimpl-2
```
package server-linked-1 where
  include partial-server
  include socketimpl-1

package server-linked-2 where
  include partial-server
  include socketimpl-2
Reuse

package server-linked-1 where
include partial-server
include socketimpl-1

package server-linked-2 where
include partial-server
include socketimpl-2

package multi where
A = {include server-linked-1}
B = {include server-linked-2}
Main = import qualified A.Server
import qualified B.Server

...
Reuse

```haskell
package server-linked-1 where
  include partial-server
  include socketimpl-1

package server-linked-2 where
  include partial-server
  include socketimpl-2

package multi where
A = {include server-linked-1}
B = {include server-linked-2}
Main = [import qualified A.Server
        import qualified B.Server
        ...]
```
package server-linked-1 where
  include partial-server
  include socketimpl-1

package server-linked-2 where
  include partial-server
  include socketimpl-2

package multi where
  A = \{include server-linked-1\}
  B = \{include server-linked-2\}
  Main = [import qualified A.Server
           import qualified B.Server]
           ...

Reuse

Socket Server
socketimpl-1 partial-server

Socket Server
socketimpl-2

A

Socket Server

B

Socket Server
package server-linked-1 where
include partial-server
include socketimpl-1

package server-linked-2 where
include partial-server
include socketimpl-2

package multi where
A = {include server-linked-1}
B = {include server-linked-2}
Main = [import qualified A.Server
import qualified B.Server
...]

Reuse
package server-linked-1 where
include partial-server
include socketimpl-1

package server-linked-2 where
include partial-server
include socketimpl-2

package multi where
A = {include server-linked-1}
B = {include server-linked-2}
Main = [import qualified A.Server
import qualified B.Server ...

Reuse
package multi where
A = \{\text{include} server-linked-1\}
B = \{\text{include} server-linked-2\}
Main = \[
\text{import qualified A.Server} \\
\text{import qualified B.Server} \\
\ldots
\]
package multi where
A = {include server-linked-1}
B = {include server-linked-2}
Main = [import qualified A.Server
import qualified B.Server
...]

package multi-shared where
C = {include partial-server; include socketimpl-1}
D = {include partial-server; include socketimpl-1}
Main = [import qualified C.Server
import qualified D.Server
...]

Shared Reuse
**Shared Reuse**

**package multi where**

A = \{include server-linked-1\}

B = \{include server-linked-2\}

Main =

- import qualified A.Server
- import qualified B.Server


**package multi-shared where**

C = \{include partial-server; include socketimpl-1\}

D = \{include partial-server; include socketimpl-1\}

Main =

- import qualified C.Server
- import qualified D.Server


---

**Diagram:**

- A: Server → Socket
- B: Server → Socket
- Main: Socket

- C: Server
- D: Server

---

**Notes:**

- Module Expressions
- Bindings
- Package Definitions
- Package Repositories
- Module Path Names
- Module Origin
- Identity Substitutions
- Identities
- Package Definitions
- Module Path Names
**Shared Reuse**

```haskell
generate package multi where
  A = {include server-linked-1}
  B = {include server-linked-2}
  Main = [import qualified A.Server
           import qualified B.Server
         ...

generate package multi-shared where
  C = {include partial-server; include socketimpl-1}
  D = {include partial-server; include socketimpl-1}
  Main = [import qualified C.Server
           import qualified D.Server
         ...
```
Shared Reuse

package multi where
A = \{include server-linked-1\}
B = \{include server-linked-2\}
Main = [import qualified A.Server;
       import qualified B.Server;
       ...]

package multi-shared where
C = \{include partial-server; include socketimpl-1\}
D = \{include partial-server; include socketimpl-1\}
Main = [import qualified C.Server;
        import qualified D.Server;
        ...]
Shared Reuse

**package multi where**

A = \{\texttt{include server-linked-1}\}

B = \{\texttt{include server-linked-2}\}

Main = [\texttt{import qualified A.Server}]

[\texttt{import qualified B.Server}]

...

**package multi-shared where**

C = \{\texttt{include partial-server}; \texttt{include socketimpl-1}\}

D = \{\texttt{include partial-server}; \texttt{include socketimpl-1}\}

Main = [\texttt{import qualified C.Server}]

[\texttt{import qualified D.Server}]

...

A

Socket

Server

B

Socket

Server

Main

C

Socket

Server

Socket

Server
package multi where
A = {include server-linked-1}
B = {include server-linked-2}
Main = [import qualified A.Server
import qualified B.Server
...]

package multi-shared where
C = {include partial-server; include socketimpl-1}
D = {include partial-server; include socketimpl-1}
Main = [import qualified C.Server
import qualified D.Server
...]

Shared Reuse
**Shared Reuse**

**package multi where**

A = {include server-linked-1}
B = {include server-linked-2}
Main = [import qualified A.Server
import qualified B.Server
...]

**package multi-shared where**

C = {include partial-server; include socketimpl-1}
D = {include partial-server; include socketimpl-1}
Main = [import qualified C.Server
import qualified D.Server
...]

![Diagram](image-url)
package multi where
A = \{\texttt{include} \texttt{server-linked-1}\}
B = \{\texttt{include} \texttt{server-linked-2}\}
Main = \left[\begin{array}{l}
\texttt{import qualified A.Server} \\
\texttt{import qualified B.Server}
\end{array}\right]

package multi-shared where
C = \{\texttt{include} \texttt{partial-server}; \texttt{include} \texttt{socketimpl-1}\}
D = \{\texttt{include} \texttt{partial-server}; \texttt{include} \texttt{socketimpl-1}\}
Main = \left[\begin{array}{l}
\texttt{import qualified C.Server} \\
\texttt{import qualified D.Server}
\end{array}\right]
**Shared Reuse**

**package multi where**

A = \{include server-linked-1\}
B = \{include server-linked-2\}
Main = [import qualified A.Server
        import qualified B.Server
        ...

**package multi-shared where**

C = \{include partial-server; include socketimpl-1\}
D = \{include partial-server; include socketimpl-1\}
Main = [import qualified C.Server
        import qualified D.Server
        ...
Shared Reuse

package multi where
A = \{include server-linked-1\}
B = \{include server-linked-2\}
Main = [import qualified A.Server
import qualified B.Server
...]

package multi-shared where
C = \{include partial-server; include socketimpl-1\}
D = \{include partial-server; include socketimpl-1\}
Main = [import qualified C.Server
import qualified D.Server
...]

A
Socket
Server

B
Socket
Server

C
Socket
Server

D
Socket
Server

Main
Recursive Linking

package ab-sigs where
A :: [S_A]
B :: [S_B]
Recursive Linking

```haskell
package ab-sigs where
  A :: \[[S_A]\]
  B :: \[[S_B]\]

package b-from-a where
  include ab-sigs
  B = [import A; ...]

package a-from-b where
  include ab-sigs
  A = [import B; ...]
```
Recursive Linking

package ab-sigs where
  A :: [S_A]
  B :: [S_B]

package b-from-a where
  include ab-sigs
  B = [import A; ...]

package a-from-b where
  include ab-sigs
  A = [import B; ...]

package ab-rec-sep where
  include a-from-b
  include b-from-a
Recursive Linking

package ab-sigs where
A :: [S_A]
B :: [S_B]

package b-from-a where
include ab-sigs
B = [import A; . . .]

package a-from-b where
include ab-sigs
A = [import B; . . .]

package ab-rec-sep where
include a-from-b
include b-from-a
Backpack Syntax

<table>
<thead>
<tr>
<th>Category</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package Names</td>
<td>$P \in \text{PkgNames}$</td>
</tr>
<tr>
<td>Module Path Names</td>
<td>$p \in \text{ModPaths}$</td>
</tr>
<tr>
<td>Package Repositories</td>
<td>$R ::= D_1, \ldots, D_n$</td>
</tr>
<tr>
<td>Package Definitions</td>
<td>$D ::= \text{package } P \ t \ 	ext{where } B_1, \ldots, B_n$</td>
</tr>
<tr>
<td>Bindings</td>
<td>$B ::= p = [M] \mid p :: [S] \mid p = p \mid \text{include } P \ t \ r$</td>
</tr>
<tr>
<td>Thinning Specs</td>
<td>$t ::= (p_1, \ldots, p_n)$</td>
</tr>
<tr>
<td>Renaming Specs</td>
<td>$r ::= \langle p_1 \mapsto p'_1, \ldots, p_n \mapsto p'_n \rangle$</td>
</tr>
<tr>
<td>Module Expressions</td>
<td>$M ::= \text{imports; exports; defns}$</td>
</tr>
<tr>
<td>Signature Expressions</td>
<td>$S ::= \text{imports; decls}$</td>
</tr>
</tbody>
</table>

**interfaces, reuse, and recursive linking $\Rightarrow$ strong modularity**
Backpack Syntax

Package Names \( P \in PkgNames \)
Module Path Names \( p \in ModPaths \)
Package Repositories \( R ::= D_1, \ldots, D_n \)
Package Definitions \( D ::= \text{package } P t \text{ where } B_1, \ldots, B_n \)
Bindings \( B ::= p = [M] | p :: [S] \mid p = p \mid \text{include } P t r \)
Thinning Specs \( t ::= (p_1, \ldots, p_n) \)
Renaming Specs \( r ::= \langle p_1 \mapsto p'_1, \ldots, p_n \mapsto p'_n \rangle \)
Module Expressions \( M ::= \text{imports}; \text{exports}; \text{defns} \)
Signature Expressions \( S ::= \text{imports}; \text{decls} \)

\textit{interfaces, reuse, and recursive linking} \implies \textit{strong modularity}

Other features:

\begin{itemize}
  \item \textit{Aliasing}: module name aliases
\end{itemize}
## Backpack Syntax

<table>
<thead>
<tr>
<th>Package Names</th>
<th>$P \in PkgNames$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Path Names</td>
<td>$p \in ModPaths$</td>
</tr>
<tr>
<td>Package Repositories</td>
<td>$R ::= D_1, \ldots, D_n$</td>
</tr>
<tr>
<td>Package Definitions</td>
<td>$D ::= \text{package } P \ t \ where B_1, \ldots, B_n$</td>
</tr>
<tr>
<td>Bindings</td>
<td>$B ::= p = [M] \</td>
</tr>
<tr>
<td>Thinning Specs</td>
<td>$t ::= (p_1, \ldots, p_n)$</td>
</tr>
<tr>
<td>Renaming Specs</td>
<td>$r ::= \langle p_1 \mapsto p'_1, \ldots, p_n \mapsto p'_n \rangle$</td>
</tr>
<tr>
<td>Module Expressions</td>
<td>$M ::= \text{imports; exports; defns}$</td>
</tr>
<tr>
<td>Signature Expressions</td>
<td>$S ::= \text{imports; decls}$</td>
</tr>
</tbody>
</table>

### interfaces, reuse, and recursive linking $\Rightarrow$ strong modularity

**Other features:**

- **Aliasing:** module name aliases
- **Thinning:** control module exposure
Backpack Syntax

Package Names \( P \in \text{PkgNames} \)
Module Path Names \( p \in \text{ModPaths} \)
Package Repositories \( R ::= D_1, \ldots, D_n \)
Package Definitions \( D ::= \text{package } P \ t \ \text{where } B_1, \ldots, B_n \)
Bindings \( B ::= p = [M] \ | \ p :: [S] \ | \ p = p \ | \ \text{include } P \ \ t \ r \)
Thinning Specs \( t ::= (p_1, \ldots, p_n) \)
Renaming Specs \( r ::= \{ p_1 \leftarrow p'_1, \ldots, p_n \leftarrow p'_n \} \)

Module Expressions \( M ::= \text{imports; exports; defns} \)
Signature Expressions \( S ::= \text{imports; decls} \)

\textit{interfaces, reuse, and recursive linking} \( \Rightarrow \) \textit{strong modularity}

Other features:

\begin{itemize}
  \item \textbf{Aliasing}: module name aliases
  \item \textbf{Thinning}: control module exposure
  \item \textbf{Renaming}: control module naming/linking
\end{itemize}
Outline

• Language tour
• Semantics
• Future work
Goal 1) Base It on MixML

Start with MixML,* [Rossberg & Dreyer, ICFP ’08]
a minimal, fully expressive core calculus of mixin modules.
Goal 1) Base It on MixML

Start with MixML,* * [Rossberg & Dreyer, ICFP ’08]
a minimal, fully expressive core calculus of mixin modules.

but MixML
doesn’t really
work for us:
Goal 1) Base It on MixML

Start with MixML,*  * [Rossberg & Dreyer, ICFP ’08]
a minimal, fully expressive core calculus of mixin modules.

*but MixML doesn’t really work for us:

• MixML targets “LTG”
Goal 1) Base It on MixML

Start with MixML,* * [Rossberg & Dreyer, ICFP ’08]
a minimal, fully expressive core calculus of mixin modules.

* but MixML doesn’t really work for us:
  • MixML targets “LTG”
  • MixML does not support shared reuse
Goal 1) Base It on MixML

Start with MixML,* [Rossberg & Dreyer, ICFP ’08]
a minimal, fully expressive core calculus of mixin modules.

but MixML doesn’t really work for us:

• MixML targets “LTG”
• MixML does not support shared reuse
• MixML semantics is pretty complicated
Goal 2) Retrofit Haskell

Because we retrofit, we must “compile” semantics down to plain Haskell modules.
Type Representation

Main =

\[
\begin{bmatrix}
\text{import qualified A.Server} \\
\text{import qualified B.Server} \\
\text{... A.Server.ServerT ...} \\
\text{... B.Server.ServerT ...}
\end{bmatrix}
\]
Type Representation

\[
\begin{align*}
\text{Main} &= \begin{bmatrix}
\text{import qualified A.Server} \\
\text{import qualified B.Server}
\end{bmatrix} \\
&= \begin{bmatrix}
\ldots & \text{A.Server.ServerT} & \ldots \\
\ldots & \text{B.Server.ServerT} & \ldots
\end{bmatrix}
\end{align*}
\]
Type Representation

import qualified A.Server
import qualified B.Server

Main =

[... Socket ... ServerT ...

socketimpl-1 partial-server

... Socket ... ServerT ...

socketimpl-2 partial-server]
Module Identities

Identity Variables $\alpha, \beta \in \text{IdentVars}$

Identity Constructors $\mathcal{K} \in \text{IdentCtors}$

Identities $\nu ::= \alpha \mid \mathcal{K} \bar{\nu} \mid \mu\alpha.\nu$
Module Identities

Identity Variables \( \alpha, \beta \in \text{IdentVars} \)

Identity Constructors \( \mathcal{K} \in \text{IdentCtors} \)

Identities

\[ \nu ::= \alpha \mid \mathcal{K} \bar{\nu} \mid \mu \alpha.\nu \]
Module Identities

Identity Variables \( \alpha, \beta \in IdentVars \)
Identity Constructors \( \mathcal{K} \in IdentCtors \)
Identities

\[ \nu ::= \alpha \mid \mathcal{K} \nu \mid \mu\alpha.\nu \]

1) Variables for holes.
Module Identities

Identity Variables $\alpha, \beta \in IdentVars$
Identity Constructors $\mathcal{K} \in IdentCtors$
Identities $\nu ::= \alpha \mid \mathcal{K} \bar{\nu} \mid \mu \alpha.\nu$

1) Variables for holes.
Module Identities

Identity Variables \( \alpha, \beta \in \text{IdentVars} \)
Identity Constructors \( \kappa \in \text{IdentCtors} \)
Identities \( \nu ::= \alpha \mid \kappa \bar{\nu} \mid \mu \alpha . \nu \)

1) Variables for holes.
2) Fresh token applied to imported mod idents.
Module Identities

Identity Variables \( \alpha, \beta \in \text{IdentVars} \)
Identity Constructors \( \mathcal{K} \in \text{IdentCtors} \)
Identities

\[
\nu ::= \alpha | \mathcal{K} \bar{\nu} | \mu \alpha.\nu
\]

1) Variables for holes.
2) Fresh token applied to imported mod idents.
Module Identities

Identity Variables \( \alpha, \beta \in IdentVars \)
Identity Constructors \( \mathcal{K} \in IdentCtors \)
Identities \( \nu ::= \alpha | \mathcal{K} \bar{\nu} | \mu \alpha . \nu \)

1) Variables for holes.
2) Fresh token applied to imported mod idents.
3) Linking is substitution.
Module Identities

Identity Variables \( \alpha, \beta \in Ident\text{Vars} \)
Identity Constructors \( K \in Ident\text{Ctors} \)
Identities \( \nu ::= \alpha \mid K \bar{\nu} \mid \mu\alpha.\nu \)

1) Variables for holes.
2) Fresh token applied to imported mod idents.
3) Linking is substitution.
Module Identities

Identity Variables \( \alpha, \beta \in \text{IdentVars} \)
Identity Constructors \( K \in \text{IdentCtors} \)
Identities \( \nu ::= \alpha \mid K \bar{\nu} \mid \mu \alpha.\nu \)

1) Variables for holes.
2) Fresh token applied to imported mod idents.
3) Linking is substitution.
Module Identities

Identity Variables \( \alpha, \beta \in \text{IdentVars} \)
Identity Constructors \( \mathcal{K} \in \text{IdentCtors} \)
Identities

1) Variables for holes.
2) Fresh token applied to imported mod idents.
3) Linking is substitution.
4) \( \mu \)-binders for cyclic graphs (recursion).
Module Identities

Identity Variables \( \alpha, \beta \in \text{IdentVars} \)
Identity Constructors \( \mathcal{K} \in \text{IdentCtors} \)
Identities \( \nu ::= \alpha \mid \mathcal{K} \bar{\nu} \mid \mu \alpha.\nu \)

1) Variables for holes.
2) Fresh token applied to imported mod idents.
3) Linking is substitution.
4) \( \mu \)-binders for cyclic graphs (recursion).
Module Identities

Identity Variables \( \alpha, \beta \in IdentVars \)
Identity Constructors \( \mathcal{K} \in IdentCtors \)
Identities \( \nu ::= \alpha \mid \mathcal{K} \bar{\nu} \mid \mu\alpha.\nu \)

1) Variables for holes.
2) Fresh token applied to imported mod idents.
3) Linking is substitution.
4) \( \mu \)-binders for cyclic graphs (recursion).
Module Identities

Identity Variables \( \alpha, \beta \in \text{IdentVars} \)
Identity Constructors \( \mathcal{K} \in \text{IdentCtors} \)
Identities \( \nu ::= \alpha | \mathcal{K} \bar{\nu} | \mu\alpha.\nu \)

1) Variables for holes.
2) Fresh token applied to imported mod idents.
3) Linking is substitution.
4) \( \mu \)-binders for cyclic graphs (recursion).

module identities enable shared reuse and recursive linking!
module identity

Type System
module identity

Type System

Elaboration
Type System
(two passes on a package)

I) Shaping

- Determine structure of modules and code
- Synthesize module identities
Type System

(two passes on a package)

1) Shaping
   • Determine structure of modules and code
   • Synthesize module identities

2) Typing
   • Do Haskell typechecking
   • Augment shapes with Haskell typing information
Type System: Linking

\[
\Delta \vdash B_1 \Rightarrow \tilde{\Xi}_1 \quad \Delta; \tilde{\Xi}_1 \vdash B_2 \Rightarrow \tilde{\Xi}_2 \quad \vdash \tilde{\Xi}_1 + \tilde{\Xi}_2 \Rightarrow \tilde{\Xi} \quad (\text{SHSEQ})
\]

\[
\Delta; \tilde{\Xi}_{\text{pkg}} \vdash B_1 : \Xi_1 \quad \Delta; \Xi_1; \tilde{\Xi}_{\text{pkg}} \vdash B_2 : \Xi_2 \quad \Xi_1 \oplus \Xi_2 = \Xi \text{ defined} \quad (\text{TYSEQ})
\]

\[
\Delta; \tilde{\Xi}_{\text{pkg}} \vdash B_1, B_2 : \Xi
\]
Type System: Linking

$$\Delta \vdash B_1 \Rightarrow \tilde{\Xi}_1 \quad \Delta; \tilde{\Xi}_1 \vdash B_2 \Rightarrow \tilde{\Xi}_2 \quad \vdash \tilde{\Xi}_1 + \tilde{\Xi}_2 \Rightarrow \tilde{\Xi}$$  (SHSEQ)

$$\Delta; \tilde{\Xi}_{pkg} \vdash B_1 : \Xi_1 \quad \Delta; \Xi_1; \tilde{\Xi}_{pkg} \vdash B_2 : \Xi_2 \quad \Xi_1 \oplus \Xi_2 = \Xi \text{ defined}$$  (TYSEQ)
Type System: Linking

\[ \Delta \vdash B_1 \Rightarrow \tilde{\Xi}_1 \]
\[ \Delta; \tilde{\Xi}_1 \vdash B_2 \Rightarrow \tilde{\Xi}_2 \quad \vdash \tilde{\Xi}_1 + \tilde{\Xi}_2 \Rightarrow \tilde{\Xi} \quad (\text{SHSEQ}) \]
\[ \Delta \vdash B_1, B_2 \Rightarrow \tilde{\Xi} \]
\[ \Delta; \tilde{\Xi}_\text{pkg} \vdash B_1 : \Xi_1 \]
\[ \Delta; \Xi_1; \tilde{\Xi}_\text{pkg} \vdash B_2 : \Xi_2 \]
\[ \Xi_1 \oplus \Xi_2 = \Xi \text{ defined} \quad (\text{TYSEQ}) \]
\[ \Delta; \tilde{\Xi}_\text{pkg} \vdash B_1, B_2 : \Xi \]
Type System: Linking

\[
\begin{align*}
\Delta \vdash B_1 &\Rightarrow \Xi_1 \\
\Delta; \Xi_1 \vdash B_2 &\Rightarrow \Xi_2 \\
\vdash \Xi_1 + \Xi_2 &\Rightarrow \Xi
\end{align*}
\]  
\hspace{5cm} \text{(SHSEQ)}

\[
\begin{align*}
\Delta; \Xi_{pkg} \vdash B_1 : \Xi_1 \\
\Delta; \Xi_1; \Xi_{pkg} \vdash B_2 : \Xi_2 \\
\Xi_1 \oplus \Xi_2 &\text{ defined}
\end{align*}
\]  
\hspace{5cm} \text{(TYSEQ)}

\[
\Delta; \Xi_{pkg} \vdash B_1, B_2 : \Xi
\]
Type System: Linking

\[
\frac{\Delta \vdash B \Rightarrow \tilde{\Xi}_1 \quad \Delta; \tilde{\Xi}_1 \vdash B_2 \Rightarrow \tilde{\Xi}_2}{\Delta \vdash B_1, B_2 \Rightarrow \tilde{\Xi}} \quad \text{unification!} \\
\frac{\Delta; \tilde{\Xi}_{pkg} \vdash B_1 : \Xi_1 \quad \Delta; \Xi_1; \tilde{\Xi}_{pkg} \vdash B_2 : \Xi_2}{\Delta; \tilde{\Xi}_{pkg} \vdash B_1, B_2 : \Xi} \quad \Xi_1 + \Xi_2 = \Xi \text{ defined} \quad (\text{SHSEQ})
\]
Type System: Linking

\[
\Delta \vdash B_1 \Rightarrow \Xi_1 \quad \Delta; \Xi_1 \vdash B_2 \Rightarrow \Xi_2 \quad \vdash \Xi_1 + \Xi_2 \Rightarrow \Xi \\
\Delta \vdash B_1, B_2 \Rightarrow \Xi \quad \text{(SHSEQ)}
\]

\[
\Delta; \Xi_{pkg} \vdash B_1 : \Xi_1 \quad \Delta; \Xi_1; \Xi_{pkg} \vdash B_2 : \Xi_2 \quad \Xi_1 \oplus \Xi_2 = \Xi \text{ defined} \quad (\text{TYSEQ})
\]

\[
\Delta; \Xi_{pkg} \vdash B_1, B_2 : \Xi
\]
Type System: Linking

\[
\Delta \vdash B_1 \Rightarrow \tilde{\Xi}_1 \\
\Delta; \tilde{\Xi}_1 \vdash B_2 \Rightarrow \tilde{\Xi}_2 \\
\vdash \tilde{\Xi}_1 + \tilde{\Xi}_2 \Rightarrow \tilde{\Xi}
\]

\[
\Delta; \tilde{\Xi}_{pkg} \vdash B_1 : \Xi_1 \\
\Delta; \Xi_1; \tilde{\Xi}_{pkg} \vdash B_2 : \Xi_2 \\
\Xi_1 \oplus \Xi_2 = \Xi \text{ defined}
\]

\[
\Delta; \tilde{\Xi}_{pkg} \vdash B_1, B_2 : \Xi
\]
Type System: Linking

\[
\begin{align*}
\Delta \vdash B_1 &\Rightarrow \Xi_1 & \Delta; \Xi_1 \vdash B_2 &\Rightarrow \Xi_2 & \vdash \Xi_1 + \Xi_2 &\Rightarrow \Xi \quad \text{(SHSEQ)} \\
\Delta \vdash B_1, B_2 &\Rightarrow \Xi \\
\Delta; \Xi_{pkg} \vdash B_1 : \Xi_1 & \quad \Delta; \Xi_1; \Xi_{pkg} \vdash B_2 : \Xi_2 & \quad \Xi_1 \oplus \Xi_2 = \Xi \text{ defined} & \quad \Delta; \Xi_{pkg} \vdash B_1, B_2 : \Xi \quad \text{(TYSEQ)}
\end{align*}
\]

Backpack

MixML

\[
\begin{align*}
\Gamma; \mathcal{R} \cup \mathcal{R}_1 \cup \mathcal{L}_1; \overline{\beta_1} &\vdash \text{mod}_1 : \Sigma_1 \\
\Gamma; \mathcal{R} \cup \mathcal{R}_2 \cup \mathcal{L}_2; \overline{\beta_2} &\vdash \text{stat} \text{mod}_2 : \Sigma'_2 \\
\Gamma; \mathcal{R} \cup \mathcal{R}_2 \cup \delta \mathcal{L}_2; \overline{\beta_2} &\vdash \text{mod}_2 : \Sigma_2 \\
\Gamma; \mathcal{R} \cup \mathcal{R}_2; \overline{\beta_1}, \overline{\beta_2} &\vdash (X = \text{mod}_1) \text{ with } \text{mod}_2 : \Sigma \quad \text{(LINK)}
\end{align*}
\]
Elaboration

module $\mathcal{K}_{\text{Sock}}$ where
  data SockT = ...

module $\mathcal{K}^\text{alt}_{\text{Sock}}$ where
  data SockT = ...

module $\mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}})$ where
  import $\mathcal{K}_{\text{Sock}}$ as Socket(SockT)
  data ServerT = ...SockT...

module $\mathcal{K}_{\text{Ser}}(\mathcal{K}^\text{alt}_{\text{Sock}})$ where
  import $\mathcal{K}^\text{alt}_{\text{Sock}}$ as Socket(SockT)
  data ServerT = ...SockT...

module $\mathcal{K}_{\text{Main}}(\mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}), \mathcal{K}_{\text{Ser}}(\mathcal{K}^\text{alt}_{\text{Sock}}))$ where
  import qualified $\mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}})$ as A.Server(ServerT)
  import qualified $\mathcal{K}_{\text{Ser}}(\mathcal{K}^\text{alt}_{\text{Sock}})$ as B.Server(ServerT)
  ...

...
Elaboration

- One module file per identity

```
module \( \mathcal{K}_{\text{Sock}} \) where
  data SockT = ...

module \( \mathcal{K}_{\text{Sock}}^{\text{alt}} \) where
  data SockT = ...

module \( \mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}) \) where
  import \( \mathcal{K}_{\text{Sock}} \) as Socket(SockT)
  data ServerT = ...SockT...

module \( \mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}^{\text{alt}}) \) where
  import \( \mathcal{K}_{\text{Sock}}^{\text{alt}} \) as Socket(SockT)
  data ServerT = ...SockT...

module \( \mathcal{K}_{\text{Main}}(\mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}), \mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}^{\text{alt}})) \) where
  import qualified \( \mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}) \) as A.Server(ServerT)
  import qualified \( \mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}^{\text{alt}}) \) as B.Server(ServerT)
  ...
```
Elaboration

- One module file per identity
- Code duplicated (a la C++ templates)

```
module K_Sock where
  data SockT = ... 

module K_Sock^alt where
  data SockT = ... 

module K_Ser(K_Sock) where
  import K_Sock as Socket(SockT)
  data ServerT = ...SockT...

module K_Ser(K_Sock^alt) where
  import K_Sock^alt as Socket(SockT)
  data ServerT = ...SockT...

module K_Main(K_Ser(K_Sock), K_Ser(K_Sock^alt)) where
  import qualified K_Ser(K_Sock) as A.Server(ServerT)
  import qualified K_Ser(K_Sock^alt) as B.Server(ServerT)
  ...
```
Elaboration

- One module file per identity
- Code duplicated (a la C++ templates)
- Code largely preserved

```haskell
module \( C_{\text{Sock}} \) where
  data SockT = ...
data ServerT = ...SockT...

module \( C_{\text{alt Sock}} \) where
  data SockT = ...

module \( C_{\text{Ser}}(C_{\text{Sock}}) \) where
  import \( C_{\text{Sock}} \) as Socket(SockT)
data ServerT = ...SockT...

module \( C_{\text{Ser}}(C_{\text{alt Sock}}) \) where
  import \( C_{\text{alt Sock}} \) as Socket(SockT)
data ServerT = ...SockT...

module \( C_{\text{Main}}(C_{\text{Ser}}(C_{\text{Sock}}), C_{\text{Ser}}(C_{\text{alt Sock}})) \) where
  import qualified \( C_{\text{Ser}}(C_{\text{Sock}}) \) as A.Server(ServerT)
  import qualified \( C_{\text{Ser}}(C_{\text{alt Sock}}) \) as B.Server(ServerT)
  ...
```
Elaboration

- One module file per identity
- Code duplicated (a la C++ templates)
- Code largely preserved
- Rewrite imports to identities
Elaboration

- One module file per identity
- Code duplicated (a la C++ templates)
- Code largely preserved
- Rewrite imports to identities

\[
\begin{align*}
\text{module } & \mathcal{K}_{\text{Sock}} \text{ where} \\
& \text{data } \text{SockT} = \ldots
\end{align*}
\]

\[
\begin{align*}
\text{module } & \mathcal{K}_{\text{alt}} \text{ where} \\
& \text{data } \text{SockT} = \ldots
\end{align*}
\]

\[
\begin{align*}
\text{module } & \mathcal{K}_{\text{Ser}(\mathcal{K}_{\text{Sock}})} \text{ where} \\
& \text{import } \mathcal{K}_{\text{Sock}} \text{ as } \text{Socket(SockT)} \\
& \text{data } \text{ServerT} = \ldots \text{SockT} \\
& \text{import } \mathcal{K}_{\text{alt}} \text{ as } \text{Socket(SockT)} \\
& \text{data } \text{ServerT} = \ldots \text{SockT}
\end{align*}
\]

\[
\begin{align*}
\text{module } & \mathcal{K}_{\text{Main}(\mathcal{K}_{\text{Ser}(\mathcal{K}_{\text{Sock}})}, \mathcal{K}_{\text{Ser}(\mathcal{K}_{\text{alt}})})} \text{ where} \\
& \text{import qualified } \mathcal{K}_{\text{Ser}(\mathcal{K}_{\text{Sock}})} \text{ as } \text{A.Server(ServerT)} \\
& \text{import qualified } \mathcal{K}_{\text{Ser}(\mathcal{K}_{\text{alt}})} \text{ as } \text{B.Server(ServerT)} \\
& \ldots
\end{align*}
\]
Elaboration

- One module file per identity
- Code duplicated (a la C++ templates)
- Code largely preserved
- Rewrite imports to identities

```
module  \( K_{\text{Sock}} \) where
  data SocketT = ...

module  \( K_{\text{Sock}}^{\text{alt}} \) where
  data SocketT = ...

module  \( K_{\text{Ser}}(K_{\text{Sock}}) \) where
  import \( K_{\text{Sock}} \) as Socket(SocketT)
  data ServerT = ...SocketT...

module  \( K_{\text{Ser}}(K_{\text{Sock}}^{\text{alt}}) \) where
  import \( K_{\text{Sock}}^{\text{alt}} \) as Socket(SocketT)
  data ServerT = ...SocketT...

module  \( K_{\text{Main}}(K_{\text{Ser}}(K_{\text{Sock}}), K_{\text{Ser}}(K_{\text{Sock}}^{\text{alt}})) \) where
  import qualified \( K_{\text{Ser}}(K_{\text{Sock}}) \) as A.Server(ServerT)
  import qualified \( K_{\text{Ser}}(K_{\text{Sock}}^{\text{alt}}) \) as B.Server(ServerT)
  ...
```
Elaboration

- One module file per identity
- Code duplicated (a la C++ templates)
- Code largely preserved
- Rewrite imports to identities

module \( \mathcal{K}_{\text{Sock}} \) where
  data SockT = ...

module \( \mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}) \) where
  import \( \mathcal{K}_{\text{Sock}} \) as Socket(SockT)
  data ServerT = ...SockT...

module \( \mathcal{K}_{\text{Main}}(\mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}), \mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}})) \) where
  import qualified \( \mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}) \) as C.Server(ServerT)
  import qualified \( \mathcal{K}_{\text{Ser}}(\mathcal{K}_{\text{Sock}}) \) as D.Server(ServerT)
  ...

Code largely preserved
Type System

module identity simplifies and organizes the semantics
module identity simplifies and organizes the semantics
What else is in the paper?

- Thinning and renaming
- Formalization of Haskell modules
- Elaboration soundness statement & proof
- Metatheory/axioms for “core language”
What else is in the paper?

• Thinning and renaming
• Formalization of Haskell modules
• Elaboration soundness statement & proof
• Metatheory/axioms for “core language”

53-page appendix
Future Work

type classes & integration with Cabal package manager
Contributions

- Retrofits Haskell with strong modularity
  - Designed at package level
  - Employs simplified *mixin* design
  - Defined as *elaboration* into weak Haskell modules
  - Separate *typechecking*, not separate compilation

- Generic design could work for other weak langs
Targeting an existing, weak module language is not just practical but *interesting*!
Targeting an existing, weak module language is not just practical but *interesting*!

Thanks!
Bonus Materials
Proving soundness was hard.

• First had to formally define “plain Haskell modules” and binary interface files
  - But not Haskell typechecking!

• Only formalization* of Haskell modules with:
  - separately checkable recursive modules
  - conventional metatheory, e.g. *Weakening*, *Substitution*

• Definitions and metatheory are parameterized by axioms for “core” level of terms and types
  - Could generalize to other “weak” langs with Haskell’s module semantics

* others: [Faxén, JFP ’02] [Diatchki et al., Haskell ’02]
Type Classes

```haskell
package problematic where

A :: [data T; ...]
B :: [data T; ...]

C = import qualified A
 import qualified B

instance Eq A.T where
eq t1 t2 = True

instance Eq B.T where
eq t1 t2 = False
```

- Which instances from A and B impls does C see?
  - Elab. of C must specify imported instances

- Package type presumes A.T distinct from B.T
  - Can’t use same impl for A and for B
Implementation Check

• Author wants to check that she implements sig
• But not all holes in sig are implemented in impl
• How exactly to define implements then?

package containers-sig-1.x where
    include prelude-sig-4.x
Data.Set :: [import Prelude; ...]

package containers-impl-1.5 where
    include prelude-sig-4.x
HiddenImpl = [...] [import Prelude]
Data.Set = [...] [import HiddenImpl]

implements containers-sig-1.x
Conditionals

```haskell
package mylib where

Foo =

#ifdef __GHC__
import GhcStuff
...ghc-specific impl...
#else
...default impl...
#endif
```

```haskell
package mylib where

Foo =

case compiler of

ghc ⇒ import GhcStuff
...ghc-specific impl...

_ ⇒ [...default impl...]
```