In submission

Backpack to Work: Towards Practical Mixin Linking for Haskell

Edward Yang

Stanford University

Abstract

Scott Kilpatrick
MPI-SWS

Derek Dreyer
MPI-SWS

Simon Peyton Jones
Microsoft Research

Thus motivated, we describe how to divide Backpack'14 across
In this paper, we describe an evolution of the Backpack mixin package system which respects the division between package manager

and compiler in the Haskell ecosystem: Backpack'16. Programs written in Backpack'16 are processed in two phases: first, a mixin linking phase which computes a "wiring diagram" of components indifferent to the actual Haskell source code, and then a typechecking phase on the output of mixin linking which processes Haskell source. This is not merely a paper design: our architecture was principally motivated by our experiences implementing Backpack'16 in the GHC compiler and the Cabal package system.

1. Introduction

The universal organizing principle for large software systems in programming languages today is the *package*, the unit by which reusable code may be versioned and distributed. However, most package systems provide only a weak form of modularity, where packages depend directly on other packages. A stronger form of modularity would support *separate modular development*, where a package may be typechecked against an *interface* of its dependency. While such facilities have been implemented as extensions to the core language (e.g., the ML module system), these extensions say little about modularity in the *package language*, which is generally independent from the core language and implemented by a separate tool.

The Backpack package system [7] (hereafter called Backpack'14) broke new ground, arguing that *mixin packages* could be a good fit for providing package-level modularity. Mixin packages consist of defined modules (provisions) and declared signatures (requirements); these packages can be combined together through a process of mixin linking, which wires up provisions with requirements. Mixin packages fit very naturally into the existing patterns of use for packages. Moreover, since they extend only the pack-

age language they can be retrofitted onto Haskell without requiring "yet another type system extension." Finally, mixin linking helps avoid the preponderance of *sharing constraints* and the so-called "fully-functorized" style commonly associated with ML functors.

There is just one problem: Backpack'14 was never implemented. Worse, it cannot be implemented—at least, not as an extension to the package manager. The problem is that the semantics of Backpack'14 were closely entwined with the semantics of Haskell itself. It did not respect the traditional abstraction barrier between the compiler and package manager—a direct implementation would require close coupling between the two.

the abstraction barrier between the package manager and the compiler; we call our system Backpack'16. We consider separately the problem of *mixin linking*, which is indifferent to Haskell source code, and the problem of *typechecking against interfaces*, which is purely the concern of the compiler. To focus the paper, we do not address mutual recursion between packages, allowing us to avoid some orthogonal technical complications. We believe that in practice this is a minor limitation.

Specifically, our contributions are as follows:

- On the package manager side, we describe the new, unordered package language users program in Section 2, and show how to *mixin link* this language while being indifferent to the contents of Haskell source files (Sections 3.2 and 4). We achieve this by recasting mixin linking as the process of computing a *wiring diagram* which describes how the requirements of depended upon components are *instantiated*. In fact, the algorithm is essentially

equivalent to early descriptions of mixin linking in the literature; the difference is that the structure of these wiring diagrams serve as the basis for type equality in the compiler. Mixin linking produces an intermediate representation, the *mixed component language*, which mediates between the compiler and the package manager.

- On the compiler side, we describe how to typecheck a mixed component when its requirements are unfilled (Section 3.4) and how to subsequently compile it when it is instantiated (Section 3.5). These operations require only modest changes to GHC, specifically the ability to compute the type of an instantiated component. We provide a declarative typing judgment for mixed components, acting as a specification for the implementation of the former process. (Section 5).
- We have implemented our design in the GHC compiler and the Cabal package manager.[1] Though the true test of Backpack'16 would be "Backpack"ing in the wild and seeing how it can be used, we give some evidence that Backpack'16 works by Backpack'ifying two substantial, real world Haskell packages, alongside a menagarie of synthetic test cases written in alternate implementation of mixin linking which is implemented entirely by the compiler (Section 6).

Although the idea of separating the core language and the module/package language is not novel [9, 12, 14], we are first to ex-

[1] See PC chair for link to open source release.

ploit these ideas in service of an actual implementation, retrofitting strong modularity to an existing programming language. Some significant tasks remain to complete Backpack'16 (Section 7), but we have enough experience working with Backpack'16 that we believe that this design is practical.

2. A tour of Backpack'16

In this section, we give a "user's eye" tour of the extensions to Cabal and GHC which constitute Backpack'16. While Backpack'16 is in many respects similar to Backpack'14 [7], we will not assume familiarity with Backpack or Cabal; however, we will pay close attention to aspects of the design which facilitate the split of Backpack'16 across the package manager and compiler.

2.1 The current state of affairs

A *package* is the unit of distribution, versioning, and sharing, and a *package manager* allows developers to reuse code written by other people. In the case of Haskell, Cabal is the package manager for GHC and Hackage is the site where Cabal packages are uploaded and indexed. Each Cabal package has a *Cabal file*, which describes one or more *components* (libraries, executables, test-suites, etc.), each of which defines modules (with source code in separate files, ommitted from our examples) and specifies dependencies on other packages. Here are four example Cabal files, each describing a single component[2]:

```
name: base
library
```

```
    exposed-modules: Prelude

  name: bytestring
  library
    build-depends: base
    exposed-modules: Data.ByteString

  name: mysql
  library
    build-depends: base, bytestring
    exposed-modules: Db.MySQL

  name: mysql-dsl
  library
    build-depends: base, mysql, bytestring
    exposed-modules: DSL.MySQL
```

Given a package description, Cabal automates the process of retrieving its dependencies, determining a build order, interfacing with GHC to build them and installing the results.[3]

Status quo "modularity" via versions A `build-depends` field also specifies acceptable version range for a dependency:

```
  name: mysql
  version: 1.0
  library
    build-depends: base, bytestring >= 0.9
    exposed-modules: Db.MySQL
```

[2] Though a package is not synonymous with a component, they are often

conflated, since in Cabal, a dependency on a package actually specifies a dependency on the library component of the package.

[3] Strictly speaking, Cabal handles the process of building a single package, while cabal-install handles the process of building multiple packages. In practice, however, there is a lot of overlap between the functionality the two implement, so we refer to both as Cabal in this paper.

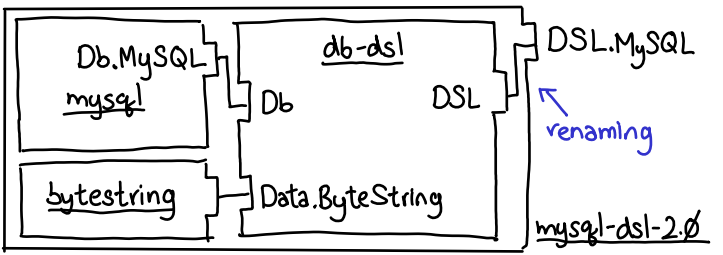


Figure 1. *Programming against an interface.* This is the wiring diagram for `mysql-dsl-2.0`. The left side of the blocks (representing components) have input ports (required signatures), while the right hand side have output ports (provided modules). Ports are wired up to show how requirements are filled; a kink indicates that some renaming took place. Mixin linking wires up requirements and provisions which have the same module name.

```
name: mysql-dsl
version: 1.0
```

```
library
  build-depends:
    base, mysql == 1.*, bytestring >= 0.10
  exposed-modules: DSL.MySQL
```

Thus, a `build-depends` constraint is not a completely definite reference: the specific version of a dependency is chosen by Cabal's *dependency resolution* step, allowing libraries to be reused at different versions of their dependencies.

2.2 Programming against an interface

The `build-depends` constraints have the limitation of committing to a specific library (albeit not a specific version) to implement a dependency. Perhaps this is fine for `base`, but we may not want to force `mysql` on the users of `mysql-dsl` —they should be able to use our library with any database that implements the required interface. In Backpack'16, a direct dependency can be replaced with a *required signature*:

```
name: db-dsl
library
  build-depends: base
  required-signatures: Data.ByteString, Db
  exposed-modules: DSL
```

Two new files, `Db.hsig` and `Data/ByteString.hsig`, accompany the package description and give the signatures of the functions used by `db-dsl`. A library with requirements is *uninstantiated*: it can be typechecked but not run. Every signature source file specifies the types, instances, functions, values, etc., which must be

provided by any module implementing this requirement. For example, the source for `Db.hsig` might be:

```
signature Db where
  import Prelude (IO)
  import Data.ByteString (ByteString) -- a sig!
  data Connection
  run :: Connection -> ByteString -> IO ()
```

A client can instantiate a requirement merely by having a module in scope with the same module name as the requirement; a process called *mixin linking* instantiates the requirement. Below, we instantiate `db-dsl` with `mysql` and `bytestring` (depicted in Figure 1):

```
name: mysql-dsl
version: 2.0
library
```

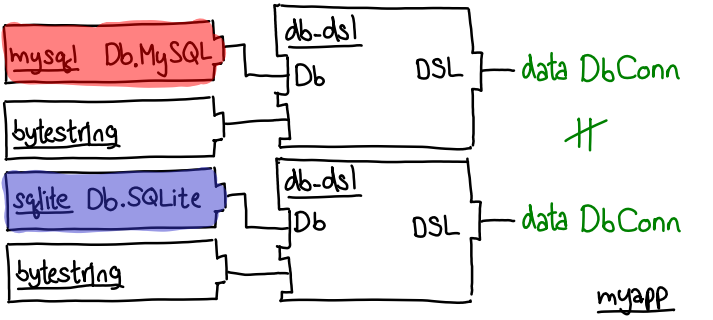


Figure 2. *Reusing libraries like functors.* Here, we consider two instantiations of `db-dsl` which define `data DbConn`. The two `DbConn`s are not type equivalent, because the respective wiring diagrams they are associated with are not equal.

```
build-depends: db-dsl, mysql, bytestring
backpack-includes: mysql (Db.MySQL as Db)
reexported-modules: DSL as DSL.MySQL
```

`db-dsl`'s requirements `Data.ByteString` and `Db` are filled by the modules provided by packages `bytestring` and `mysql`, respectively. Note that in the case of `mysql`, the module has the wrong name, so we use the `backpack-includes`[4] to rename it to the correct name. Additionally, the `reexported-modules` line reexports `db-dsl`'s module under a new name `DSL.MySQL`.

A practical consideration is whether or not there is any performance cost to using signatures. In particular, if one separately compiles uninstantiated components to machine code, no cross-package

inlining can occur, since the component is compiled only against a signature and not against the code that implements the signature. For Haskell and GHC, cross-module inlining is a major contributor to performance, so Backpack'16 does *not* separately compile uninstantiated packages. Instead, every distinct instantiation of a component is compiled against the code that implements its requirements. This process is managed by Cabal to avoid unnecessary recompilation, similarly to how Cabal avoids recompiling dependencies that are already installed.

2.3 Reusing libraries with different instantiations

Suppose an application wants to use `db-dsl` with two different databases at the same time. It can do so by mentioning `db-dsl` twice in `backpack-includes` (depicted in Figure 2):

```
name: myapp
library
  build-depends:
    db-dsl, mysql, sqlite, bytestring
  backpack-includes:
    db-dsl (DSL as DSL.MySQL)
      requires (Db as Db.MySQL)
    db-dsl (DSL as DSL.SQLite)
      requires (Db as Db.SQLite)
  exposed-modules: App
```

This gives us two copies of `db-dsl`: one with `Db` filled with `Db.MySQL`, and one with `Db` filled with `Db.SQLite`.

However, an important subtlety arises in this situation. Ordinarily, two types are considered equivalent if they have the same *identity*: an identity consists of both the name of the type and the name

of the module which originally defined the type. Suppose, however, that `db-dsl` contained a module which defined a type:

[4] Cabal already uses `includes` to denote C header includes.

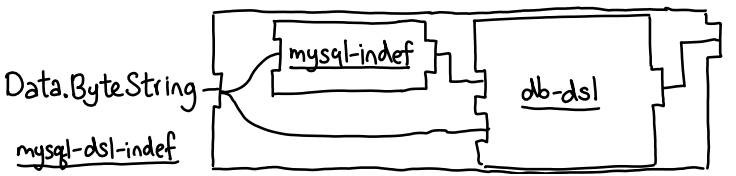


Figure 3. *Composing libraries with requirements.* It is possible to use components without filling all of their requirements; in that case, those requirements are propagated to the requirements of the enclosing component.

```
module DSL(DbConn) where
  import Db
  data DbConn = DbConn Connection URL
```

The identity of `DbConn` must somehow depend on how we decided to implement `Connection`: the `DbConn` backed by `mysql` is distinct from the `DbConn` backed by `sqlite` (after all, they may contain utterly different `Connection` values). Thus, in Backpack'16, the identity of a type also contains the *wiring* diagrams of the components which defined the types in question. Now, the two `DbConn`s

come from distinct `db-dsl`s which have different wiring diagrams; thus, they are distinct types in Haskell.

2.4 Composing libraries with requirements

Unlike functors, it is easy to use mixin linking to only partially instantiate components or combine their requirements. For example, consider this alternate version of `mysql` with a requirement:

```
name: mysql-indef
library
  build-depends: base
  required-signatures: Data.ByteString
  exposed-modules: Db.MySQL
```

One might like to reformulate `mysql` like this so that it can be used with any package implemeting the `Data.ByteString` interface. We can use this package to create a version of `db-dsl` which is partially instantiated: that is, it is instantiated with `mysql-indef`, but leaves `Data.ByteString` unimplemented (depicted in Figure 3):

```
name: mysql-dsl-indef
library
  build-depends: db-dsl, mysql-indef
  backpack-includes: mysql-indef (Db.MySQL as Db)
  reexported-modules: DSL.Db as DSL.Db.MySQL
```

`mysql-dsl-indef` has an implicit requirement `Data.ByteString`, which is the *merge* of the requirements of `db-dsl` and `mysql-indef`. For example, if we have the requirements:

```
-- db-dsl's requirement
signature Data.ByteString
```

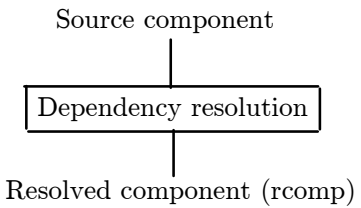
```
    data ByteString
    length :: ByteString -> Int

  -- mysql-indef's requirement
  signature Data.ByteString
    data ByteString
    concat :: [ByteString] -> ByteString
```

The merged requirement is:

```
  signature Data.ByteString
    data ByteString
    length :: ByteString -> Int
    concat :: [ByteString] -> ByteString
```


Cabal



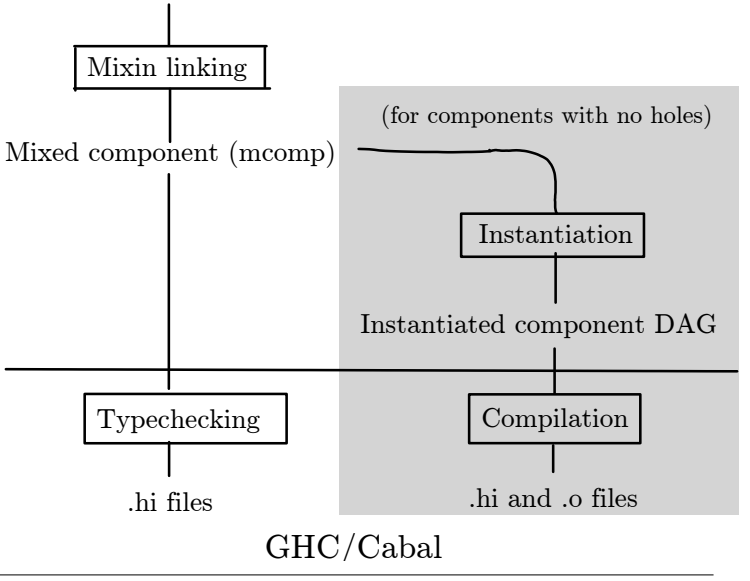


Figure 4. Diagram of the pipeline

Notice that the two type declarations for `ByteString` have been merged to one: this is how mixin linking eliminates the need for sharing constraints!

3. The pipeline

The implementation and semantics of Backpack'16 is structured as a series of passes (Figure 4) on several intermediate languages (Figure 5). The key idea is that some of these passes can be performed

solely by the package manager, without inspecting any Haskell code. In this section, we will give a detailed summary of all the phases in this pipeline, starting with the source package language described in the previous section, and continuing on to two intermediate languages that will serve as the basis for our semantic account for Backpack'16. Our running example will be a simplified version of the final example from the tour (Figure 3), shown in Figure 8.

3.1 Dependency resolution to a resolved component

The first step in processing a source component is *dependency resolution*, the existing Cabal phase that selects the versions and conditional flags of all direct dependencies of a source component (and their transitive direct dependencies as well). For example, Figure 6 shows the results of resolving a few examples from the tour, with each component renamed to a component identifier, which records the name, version, and a hash (e.g., `aaa`) recording how its direct dependencies were resolved to component identifiers.

The result of dependency resolution is a *resolved component* (Figure 7(a)) which has all conditionals and fields irrelevant to Backpack'16 eliminated, and all direct dependencies resolved to component identifiers. A resolved component is precisely the fragment of the Cabal package language which is relevant to Backpack'16, written in a form convenient for processing (e.g., each `exposed-module` is specified individually and explicitly associated with its source).

Dependency resolution in Cabal is a separate topic in its own right [4, 10], beyond the scope of this paper. Instead, we simply

Source component The smallest unit of user-written source which can have its dependencies resolved, e.g., a library, an executable or a test suite. A **source package** contains one or more source components, and is the unit of code that may be distributed (e.g., on Hackage). A source package is identified by a **package identifier** consisting of a package name and package version (e.g., `p-0.1`); a source component is in turn identified by a **source component identitifer** consisting of the package identifier with the name of the component.

Resolved component ($rcomp$ in Figure 7(a)) The output of dependency resolution on a source component. A resolved component is the input source with direct dependencies resolved to other resolved components, and Backpack'16-irrelevant Cabal fields dropped. As dependency resolution is nondeterministic, a resolved component is identified by a **component identifier** (p), which augments the source component identifier with the resolved component identifiers of the direct dependencies. (Thus, the component identifier identifies the entire transitive source a component depends on.) In Backpack'16, we can treat component identifiers as opaque strings.

Mixed component ($mcomp$ in Figure 7(b)) The output of mixin linking on a resolved component, where we have computed the "wiring diagram" for the direct dependencies of the unit, describing how requirements of the resolved component's direct dependencies have been (partially) filled. Mixin linking is deterministic; thus, a mixed component is also identified by a component identifier. A mixed component can be typechecked by the compiler.

Figure 5. Glossary of representations in Backpack'16

```
component db-dsl-1.0-aaa
 backpack-include: base-4.7-bbb (Prelude)
 required-signature: Data.ByteString
 required-signature: Db
 exposed-module: DSL.Db

component mysql-indef-1.0-ccc
 backpack-include: base-4.7-bbb (Prelude)
 required-signature: Data.ByteString
 exposed-module: Db.MySQL

component mysql-dsl-indef-1.0-ddd
 backpack-include: base-4.7-bbb (Prelude)
 backpack-include: db-dsl-1.0-aaa (DSL.Db) requires (Db)
 backpack-include: mysql-indef-1.0-ccc (Db.MySQL as Db)
 required-signature: Data.ByteString
 reexported-module: DSL.Db as DSL.Db.MySQL
```

Figure 6. Resolved components for `mysql-dsl-indef-1.0`.

m	Module name

$$
\begin{array}{rcll}
p, q & & & \text{Component identifier} \\
hsbody & & & \text{Module source} \\
hssig & & & \text{Signature source} \\
rcomp & ::= & \texttt{component}\ p\ \{\overline{rdecl}\} & \\
rdecl & ::= & \texttt{backpack-include:}\ p\ rns & \\
 & | & \texttt{exposed-module:}\ m\ \{hsbody\} & \\
 & | & \texttt{other-module:}\ m\ \{hsbody\} & \\
 & | & \texttt{reexported-module:}\ m\ \texttt{as}\ m' & \\
 & | & \texttt{required-signature:}\ m\ \{hssig\} & \\
rns & ::= & [(\overline{rn})]\ [\texttt{requires}\ (\overline{rn'})] & \\
rn & ::= & m\ \texttt{as}\ m' & \\
 & | & m &
\end{array}
$$

(a) Resolved components.

$$
\begin{array}{rcll}
\tilde{\Xi} & ::= & \forall \Theta.\ \{\tilde{\Sigma}\} & \text{Component shape} \\
\Theta & ::= & \overline{\langle m \rangle} & \text{Required module variables} \\
\tilde{\Sigma} & ::= & \overline{m \mapsto M} & \text{Provided modules} \\
\tilde{\Gamma} & ::= & \overline{p \triangleright \tilde{\Xi}} & \text{Component shape context}
\end{array}
$$

$$
\begin{array}{rcll}
M & ::= & P{:}m & \text{Module identity} \\
 & | & \langle m \rangle & \text{Module hole} \\
P & ::= & \underline{p[S]} & \text{Instantiated component identifier} \\
S & ::= & \overline{m = M} & \text{Module substitution}
\end{array}
$$

$$
\begin{array}{rcll}
mcomp & ::= & \mathsf{component}\ p\ \Theta\ \{\overline{mdecl}\} & \text{Mixed component} \\
mdecl & ::= & \mathsf{dependency}\ P\ (r) & \text{Direct dependency} \\
 & | & \mathsf{module}\ m\ \{hsbody\} & \text{Module definition} \\
 & | & \mathsf{signature}\ m\ \{hssig\} & \text{Signature definition}
\end{array}
$$

$$r \quad ::= \quad \overline{m \mapsto m'} \qquad \text{Module renaming}$$

(b) Component shapes and mixed components.

Figure 7. Syntax of intermediate languages

assume that we have a black box which produces resolved components. However, we do have to mention one thing about resolution: how are `build-depends` fields from a source component translated into `backpack-include` fields in the resolved component? For any package `p` mentioned in `build-depends` that is not mentioned in `backpack-includes`, it simply translates to `backpack-include: p`.

3.2 Mixin linking to a mixed component

The next step is *mixin linking*, which elaborates a resolved component into a *mixed component* and computes its *component shape* ($\tilde{\Xi} ::= \forall\Theta.\, \tilde{\Sigma}$). The latter describes what modules the component requires (Θ) and provides ($\tilde{\Sigma}$). A mixed component essentially describes the set of command line flags that will eventually be passed to GHC. The syntax of mixed components is given in Figure 7(b):

- dependency P (r) indicates an external component dependency, which should be brought into scope for imports according to the renaming r. P is an instantiated component identifier; a component identifier augmented with a *module substitution* $S ::= \overline{m = M}$ specifying how every requirement of the component is instantiated. (Compare to the `backpack-include` in a resolved component, which only gives a component identifier).

- module m $\{hsbody\}$ indicates that this component defines a module named m with Haskell source code $hsbody$.


```
component p                  component base
 required-signature: A        exposed-module: W
 required-signature: B
 exposed-module: Y           component q
                              required-signature: A
component r                   exposed-module: X
 required-signature: A
 backpack-include: base (W)
 backpack-include: p (Y) requires (B)
 backpack-include: q (X as B)
 reexported-module: Y as Z
```

(a) Heavily simplified version of Figure 6 to serve as the running example: `p` is `db-dsl`, `q` is `mysql-indef` and `r` is `mysql-dsl-indef`; signature `A` is `Data.ByteString` and signature `B` is `Db`.

---

component base $\triangleright$ {W $\mapsto$ base[]:W}
  module W(I(..)) {data I = MkI I; f (MkI i) = i}

component p ⟨A⟩ ⟨B⟩ $\triangleright$ {Y $\mapsto$ p[A=⟨A⟩, B=⟨B⟩]:Y}
  signature A {data J}

```
  signature B {data K}
  module Y {import A; import B; data L = MkL J K}

component q ⟨A⟩ ▷ {X ↦ q[A=⟨A⟩]:X}
  signature A (I(..)) {data I = MkI I}
  module X (I(..), K(..)) {import A; data K = MkK I}

component r ⟨A⟩ ▷ {Z ↦ p[A=⟨A⟩, B=q[A=⟨A⟩]:X]:Y}
  dependency base[] (W ↦ W)
  signature A (I(..)) {import W(I(..))}
  dependency q[A=⟨A⟩] (X ↦ B)
  dependency p[A=⟨A⟩, B=q[A=⟨A⟩]:X] (Y ↦ Y)
```

(b) Mixed components. Each component is ordered and annotated with its provided modules, as component $p\ \overline{\langle m \rangle}\ \triangleright\ \{\tilde{\Sigma}\}$.

$$\texttt{base} : \{\} \to \left\{ \texttt{W} : \begin{array}{l} \texttt{iface (base[]:W.I, base[]:W.f)} \\ \quad \texttt{data I :: * = MkI base[]:W.I} \\ \quad \texttt{f :: base[]:W.I} \to \texttt{base[]:W.I} \end{array} \right\}$$

$$\texttt{p} : \forall\langle \texttt{A} \rangle \langle \texttt{B} \rangle.\ \forall \{\texttt{A.J}\} \{\texttt{B.K}\}.$$
$$\left\{ \texttt{A} : \begin{array}{l} \texttt{iface (\{A.J\})} \\ \quad \texttt{data J :: *} \end{array}, \texttt{B} : \begin{array}{l} \texttt{iface (\{B.K\})} \\ \quad \texttt{data K :: *} \end{array} \right\} \to$$
$$\left\{ \texttt{Y} : \begin{array}{l} \texttt{iface (p[A=⟨A⟩, B=⟨B⟩]:Y.L)} \\ \quad \texttt{data L :: * = MkL \{A.J\} \{B.K\}} \end{array} \right\}$$

$$\texttt{q} : \forall\langle \texttt{A} \rangle.\ \forall \{\texttt{A.I}\}.$$
$$\left\{ \texttt{A} : \begin{array}{l} \texttt{iface (\{A.I\})} \\ \quad \cdots \end{array} \right\} \to$$

```
    { ...   data I :: * = MkI {A.I} }
    { X :   iface ({A.I}, q[A=⟨A⟩]:X.K)   }
    {       data K :: * = MkK {A.I} }

r : ∀⟨A⟩. ∀{A.J}.
    { A :   iface (base[]:W.I, {A.J}) } → {}
    {       data J :: *               }
```

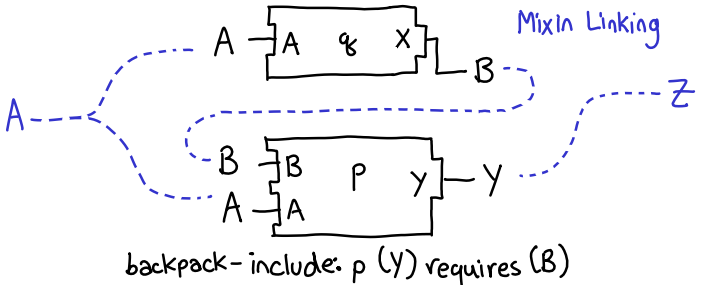
(c) Component types. The full syntax for types is in Figure 9.

Figure 8. Syntax, shapes and types for running example

- signature m $\{hssig\}$ indicates that this component has a signature named m.

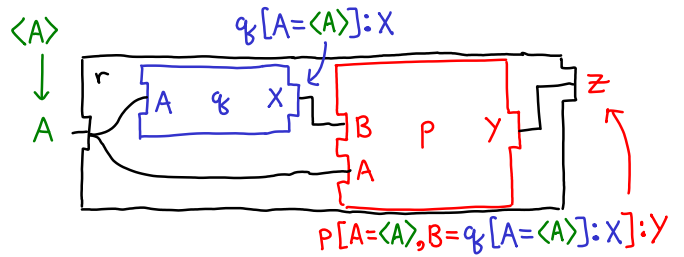
The most important part of mixin linking is the computation of instantiated component identifiers in dependency declarations. While a syntactic semantics for mixin linking is given in Section 4, the mixin linking algorithm can actually be understood entirely pictorially:

backpack-include: q (X as B)



Here, we look at the process of mixin linking `p` and `q` in the component `r` in our running example (Figure 8). The includes of `p` and `q` are represented as boxes, each of which have an input port for every required module (`A` for `q` and `A`, `B` for `p`), and an output port for every provided module (`X` for `q` and `Y` for `p`). Every port is labeled with a module name, which may be renamed according to the `backpack-include` (e.g., `X` from `q` is renamed to `B`).

These two components are linked in two ways: first, the (renamed) output of `q` is linked to the input of `p` which has the same `B`; second, the input `A` of `p` and `q` are merged together to form a single input for `r`. The result is a complete wiring diagram for `r`[5]:



With this wiring diagram in hand, we can say what the shape of `r` is by successively computing *module identities* (M) and *instantiated component identifiers* (P) for the modules and components defined in the component. In `r`, we first bind a *module variable* $\langle \texttt{A} \rangle$ (a module identity) for our unfilled requirement `A`. $\langle \texttt{A} \rangle$ feeds into the input port for `q`: thus we give `q` the instantiated component identifier $\texttt{q}[\texttt{A} = \langle \texttt{A} \rangle]$, mapping each of its input ports (`A`) to the module identity ($\langle \texttt{A} \rangle$) that filled it. As `X` is defined in `q`, it then gets the module identity $\texttt{q}[\texttt{A} = \langle \texttt{A} \rangle]:\texttt{X}$. We continue computing identities until we know the identity of every exported or reexported module in `r`; in this case, `Y` from `p` (which is reexported as `Z`). Then the component shape of `r` is $\forall \langle \texttt{A} \rangle.\, \{\texttt{Z} \mapsto \texttt{p}[\ldots]:\texttt{Y}\}$.

The elaborated mixed component for `r` can be seen in Figure 8(b); it too follows straightforwardly from the wiring diagram. In particular, the dependency requirements that are to be merged for signature `A` are from `p` and `q`, as is evident from the diagram.

3.3 Ordering a mixed component

There is a minor step which must be done before typechecking a mixed component: we must *order* the declarations of a mixed component since declarations in the earlier languages in the pipeline are

[5] Well, mostly complete; `base` is omitted since it doesn't contribute in any interesting way to linking.

Ξ	$::=$	$\forall \Theta.\, \forall \theta.\, \{\Sigma_R\} \to \{\Sigma_P\}$	Component type
θ	$::=$	$\overline{\langle m.n \rangle}$	Name variable quantifiers

$$
\begin{array}{rcll}
\Sigma & ::= & \overline{m : \tau} & \text{Provided/required types} \\
\Gamma & ::= & \overline{p : \Xi} & \text{Component typing context} \\
 & & & \\
\tau & ::= & \mathsf{iface}\ (Ns)\ \{\overline{ty}\} & \text{Module type} \\
Ns & ::= & \overline{N} & \text{Export specification} \\
ty & ::= & \texttt{data}\ n :: kind & \text{Defined entity spec} \\
 & | & \texttt{data}\ n :: kind = \cdots N \cdots & \\
 & | & n :: \cdots N \cdots & \\
 & | & \cdots & \\
 & & & \\
n & & \text{Haskell source-level entity name} & \\
N & ::= & M.n & \text{Original name} \\
 & | & \{m.n\} & \text{Name hole} \\
Sn & ::= & \overline{m.n = N} & \text{Name substitution}
\end{array}
$$

Figure 9. Semantic objects of Haskell with Backpack'16.

unordered. At this point, we step into the realm of the GHC compiler, as this ordering depends on the `import` statements within (Haskell-level) modules and signatures. We can define a topological ordering on declarations in a mixed component as follows:

- Obviously, every module/signature declaration depends on any module/signature/dependency declaration which defines a module it imports.
- Every module declaration implicitly depends on every signature declaration in the component. (This means that it is illegal for a signature to import a locally defined module. In the absence of

mutual recursion such signatures would not be implementable.)

- A dependency declaration `dependency` P depends on the signature declarations for every free module variable in P. For example, in the definition of component `r` in Figure 8(b), `signature A` must precede the dependencies on `p` and `q`, because both mention $\langle$`A`$\rangle$; but can come after the `base` dependency which does not depend on `A`.

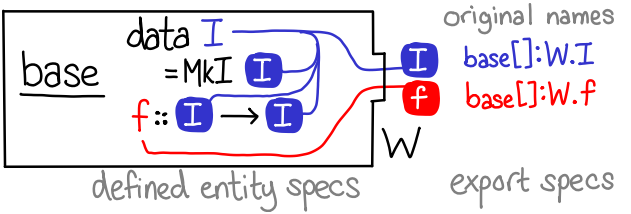
A dependency cycle indicates mutual recursion between Haskell modules and is not allowed by GHC. (In principle, other compilers might be more permissive.)

3.4 Typechecking a mixed component

After mixin linking has produced a mixed component definition from a resolved component definition, the component can now be *typechecked* into Haskell's semantic objects (Figure 9). We now consider how to typecheck each of the declarations in a mixed component. Broadly speaking, the type of a component Ξ is a function type $\Sigma_R \to \Sigma_P$, mapping from the required types Σ_R to the provided types Σ_P (we'll say more about the quantifiers shortly).

Typechecking a module A module is typechecked to produce a *module type* τ, consisting of an *export specification* Ns—what gets brought into scope when you import this module—and a set of *defined entity specifications* $\overline{ty}$—the types and values defined by this module. Syntactically, these are represented with a notation reminiscent of Haskell module declarations: `iface` (Ns) $\{\overline{ty}\}$.

For example, the module `W` in `base` (Figure 8(b)) type checks to the following module type (also given syntactically in Figure 8(c)):

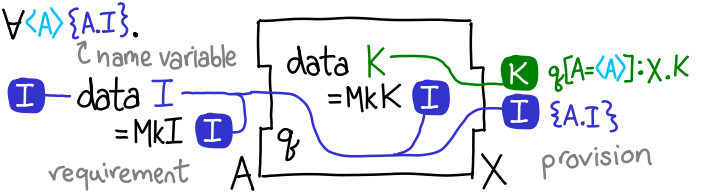


In the diagram, the enboxed references to `I` and `f` have been wired up to their definitions. In other words, there are no unresolved Haskell-source level names in a module type: all names are resolved to *original names* $N ::= M.n$, which serve as unique identifiers for any entities; in the diagram, this is shown by wiring them up. An original name for an entity defined in a module is generated by joining the identity of the module being compiled—`base[]:W` in this case— with the Haskell source level name (e.g., `I`, `f`).

Module types serve two key functions. First, the export specification denotes the entities made available when resolving a module import statement of this module.[6] Second, the defined entity specification of `base[]:W.I` denotes the "type" (or rather, the kind) of this Haskell type. If, when typechecking another module, we need to look up the kind of `base[]:W.I`, we look up the module type of `base[]:W` from the context and then find the defined entity specification for `I` within the module type.

Typechecking a signature Typechecking `q` (Figure 8(b)) from

our running example requires us to deal with a signature declaration. Here is a representation of the final type of q, which we can think of as a zoomed in version of the diagram from Section 3.2:



In the diagram, the declarations from the signature are drawn externally from q. We do not know what module will be used to fill the requirement A, and consequently, we don't know what original name will provide an implementation of I, although we do have a type which it must satisfy once we do know the original names.

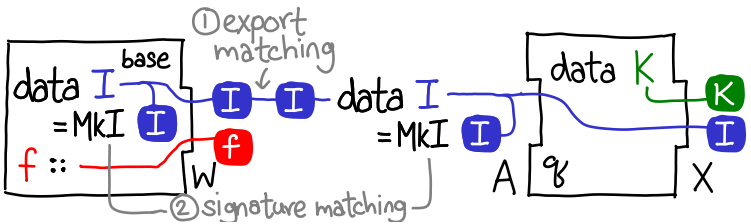
In mixin linking, we bound a module variable $\langle \mathtt{A} \rangle$ to represent the unknown module identity. In typechecking, we bind a *name variable* $\{m.n\}$ to represent the unknown entity n required by requirement m.

In theory, we typecheck a signature source file in much the same way that we a typecheck a module, assigning a fresh name variable for each defined entity in the signature. In practice, we want to avoid forcing the user to explicitly write the full requirement of the component; rather, we'd rather infer the requirement from the local signature as well as the dependencies of the component. We lack a good formal model for this process (see Section 7.2), so in the formal development in Section 5, we assume that the inferred requirement type is given nondeterministically.

You may be wondering why name variables are not simply defined as $\langle m \rangle.n$. In fact, names of this form demand that n be defined by $\langle \texttt{A} \rangle$ itself, ruling out the possibility that it *reexports* n from another module.

[6] The export specification of this module type should also mention the data constructor `MkI`. To simplify the presentation, we omit such subordinate names in export specifications. See the *espc* semantic object of Backpack'14 [7].

Typechecking a dependency Now we can define the key operation for typechecking in Backpack'16: how to compute the type of `q` instantiated with an implementation for its requirements. These instantiations are specified to the compiler via `dependency` declarations. Let's consider a simple one: filling `q`'s requirement with `W` from `base`, i.e., `dependency q[A = base[]:W]`.



There are two key steps in this diagram: first, we take the required export specification from `q`, and the provided export specification

from `base`, and we perform *export matching*, instantiating the name variable with its true identity (diagramatically, it's just a line.) Obviously, you can provide more exports than you require, but if you don't provide enough this step fails.

The second step is *signature matching*: we check if the type provided by `base` matches the required type of `q`. This step must occur after export matching: otherwise, we don't know that `Is` mentioned in the constructor `MkI` are the same.

The final types of `q` are the same types as before, but now with their module variables and name variables resolved according to the module substitution and name matching. What the final original names of `K` and `I` exported by `q`? `I` is easy: it's just `base[]:W.I`. `K` is not too difficult either; recall that the uninstantiated original name for `K` is `q[A`$=\langle$`A`$\rangle$`]:X.K`. Then, applying the module substitution `A` $=$ `base[]:W` will give you the final module name. Alternately, just read it off the diagram!

3.5 Instantiation and compilation

Typechecking is all very well and good, but what about running our programs? For both soundness reasons (Section 7) and performance reasons, it is desirable to defer compiling until we know how all the requirements of a component are to be filled.

When we have a mixed component which has no unfilled requirements, the package manager computes the *instantiated component graph*, reflecting all of the concrete code dependencies of the component. Specifically, given a closed instantiated component identity $p[S]$, for every `dependency` P of p's mixed component, recursively instantiate $P[\![S]\!]$. Furthermore, for every $m = P{:}m' \in S$, recursively instantiate P. Then, we compile each instantiated component in topological order. This compilation is ex-

actly like ordinary compilation without Backpack'16. The only new feature is that signatures compile into trivial module which reexport entities from the backing implementation; otherwise, modules that imported the signature might see entities from the backing implementation that were not exported by the signature.

It is important that the package manager computes the instantiated component graph: *applicative* semantics of instantiation mean that completely independent components could instantiate a library in the same way: in such cases we should share the compiled code. A traditional compiler cannot do this: it is solely responsible for transforming source files into object code.

4. Mixin linking

We now formalize mixin linking and describe some of its properties. In particular, we define two major operations on component shapes: $\mathsf{link}()$, which wires up the components in two shapes, and $\mathsf{rnthin}()$, which thins and renames the input and output ports of a shape; then, we say how to mixin link a resolved component.

4.1 Preliminaries

Definition 1 (Module substitutions)**.** A module substitution of the form $S ::= \overline{m = M}$ can be applied to a semantic object x with the notation $x[\![S]\!]$ (using double brackets). The application of substitution is functorial for most semantic objects, but we give some representative cases below:

$$(p[S])[\![S']\!] \quad =_P \quad p[S[\![S']\!]]$$

$$
\begin{array}{rcl}
\langle m\rangle[\![\cdot]\!] & =_M & \langle m\rangle \\
\langle m\rangle[\![m = M, S]\!] & =_M & M \\
\langle m\rangle[\![m' = M, S]\!] & =_M & \langle m\rangle[\![S]\!] \quad (m \neq m') \\
(P{:}m)[\![S]\!] & =_M & \overline{P[\![S]\!]{:}m} \\
\overline{(m = M)}[\![S]\!] & =_S & \overline{m = M[\![S]\!]} \\
(M.n)[\![S]\!] & =_N & \overline{M[\![S]\!].n} \\
\overline{(m \mapsto M)}[\![S]\!] & =_{\tilde{\Sigma}} & \overline{m \mapsto M[\![S]\!]} \\
\overline{(m : \tau)}[\![S]\!] & =_{\Sigma} & \overline{m : \tau[\![S]\!]} \\
(\mathsf{dependency}\ P\ (r))[\![S]\!] & =_{mdecl} & \mathsf{dependency}\ P[\![S]\!]\ (r) \\
(\mathsf{iface}\ (Ns)\ \{\overline{ty}\})[\![S]\!] & =_{\tau} & \mathsf{iface}\ (Ns[\![S]\!])\ \{\overline{ty[\![S]\!]}\}
\end{array}
$$

For example, $(\mathtt{p}[A = \langle \mathtt{A}\rangle]{:}\mathtt{X})[\![\mathtt{A} = \langle \mathtt{B}\rangle]\!]$ is equal to $\mathtt{p}[\mathtt{A} = \langle \mathtt{B}\rangle]{:}\mathtt{X}$. An important case where substitution does *not* occur is $\{m.n\}[\![m = \langle m'\rangle]\!]$: the result is $\{m.n\}$, not $\{m'.n\}$.

Not all syntactic component shapes are well-formed. Our definition of well-formedness rules out any "funny business":

Definition 2 (Well-formedness of component shapes)**.** A shape $\forall\Theta.\,\tilde{\Sigma}$ is well-formed if:

1. It does not provide a module variables (there is no m such that $\langle m\rangle \in \mathsf{range}(\tilde{\Sigma})$), e.g., $\forall\langle \mathtt{A}\rangle.\,\{\mathtt{B} \mapsto \langle \mathtt{A}\rangle\}$ is ill-formed,
2. It does not require what it provides (no m s.t. $\langle m\rangle \in \Theta$ and $m \in \mathsf{dom}(\tilde{\Sigma})$), e.g., $\forall\langle \mathtt{A}\rangle.\,\{\mathtt{A} \mapsto \mathtt{p}[]{:}\mathtt{A}\}$ is ill-formed,
3. It is closed; e.g., $\forall\langle \mathtt{A}\rangle.\,\{\mathtt{B} \mapsto \mathtt{p}[\mathtt{A} = \langle \mathtt{C}\rangle]{:}\mathtt{B}\}$ is ill-formed.

A well-formed shape is not necessarily well-typed.

4.2 Linking component shapes

Definition 3 (Linking on two component shapes)**.** We define $\mathsf{link}(\forall\Theta.\,\tilde{\Sigma}, \forall\Theta'.\,\tilde{\Sigma}')$, the mixin linking operation on two well-formed component shapes, as follows:

1. Unify Θ with $\tilde{\Sigma}'$, producing substitution S'. (Observe that $\mathsf{dom}(S')$ is disjoint from Θ', by property (2) of well-formedness.)
2. Unify $\tilde{\Sigma}[\![S']\!]$ with $\Theta'[\![S']\!]$, producing substitution S''.
3. Let $S = S'' \circ S'$ (the composition of substitutions.)
4. Return new substitution S and shape

$$\forall(\Theta \cup \Theta' - \mathsf{dom}(S)).\, \tilde{\Sigma}[\![S]\!] \uplus \tilde{\Sigma}'[\![S]\!]$$

Unification between Θ and $\tilde{\Sigma}$ entails unifying every $\langle m \rangle \in \Theta$ with M such that $m \mapsto M \in \tilde{\Sigma}$. $\uplus$ is a union which requires the domains of each $\tilde{\Sigma}$ to be disjoint. Unification on module identities is defined in the usual way. This operation can lifted to n-ary linking by successive linking.

Here are two useful properties about the $\mathsf{link}()$ operation:

Lemma 1 (Linking preserves well-formedness)**.** If $\tilde{\Xi}$ and $\tilde{\Xi}'$ are well-formed, then $\mathsf{link}(\tilde{\Xi}, \tilde{\Xi}')$ is well-formed.

Lemma 2 (Well-formed component shapes form a commutative monoid)**.** Let the join operation be $\mathsf{link}()$ (made total by outputting a distinguished "failure shape" in the case of error) and the bottom element be the empty shape. Then the following properties hold for all well-formed shapes:

1. Associativity: $\mathsf{link}(\mathsf{link}(\tilde{\Xi}, \tilde{\Xi}'), \tilde{\Xi}'') = \mathsf{link}(\tilde{\Xi}, \mathsf{link}(\tilde{\Xi}', \tilde{\Xi}''))$
2. Commutativity: $\mathsf{link}(\tilde{\Xi}, \tilde{\Xi}') = \mathsf{link}(\tilde{\Xi}', \tilde{\Xi})$

3. Identity: $\mathsf{link}(\tilde{\Xi}, \{\}) = \tilde{\Xi}$

The proof of associativity relies critically on the fact that it is a failure to link two component shapes which provide the same module name, even if the provided module identities happen to be the same.

4.3 Thinning and renaming component shapes

Definition 4 (Thinning and renaming component shapes)**.** We define rnthin on a well-formed shape $\tilde{\Xi}$, a total module renaming r_R and a partial module renaming r_P as follows:

$$\mathsf{rnthin}(r_R; \tilde{\Xi}; r_P) \;=\; \forall \overline{\langle r_R(m_i) \rangle}^{\,i} . \{ \overline{r_P(m'_j) \;\mapsto\; M'_j[\![r_R]\!]}^{\,j} \}$$
$$\text{where} \quad \begin{cases} \tilde{\Xi} \;=\; \forall \overline{\langle m_i \rangle}^{\,i} . \{ \overline{m'_j \;\mapsto\; M'_j}^{\,j}, \overline{m''_k \;\mapsto\; M''_k}^{\,k} \} \\ \mathsf{dom}(r_P) = \overline{m'_j}^{\,j} \end{cases}$$

Intuitively, r_R applies a renaming on all of the input ports of the component shape, while r_P applies a renaming on the output ports of a component shape. r_R must be total because we are not allowed to "drop" any requirements; furthermore when we rename a requirement we must also rename each reference to it in the provided Ms. r_P does not have to be total, and the resulting output ports of the component shape will only be those mentioned in r_P.

For example, if you have a component shaped $\forall \mathtt{B}. \{\mathtt{A} \mapsto \mathtt{p}[\mathtt{A} = \langle \mathtt{B} \rangle] : \mathtt{A}, \mathtt{C} \mapsto \mathtt{q}[] : \mathtt{C}\}$ and you apply the renaming (`A as X`) `requires` (`B as Y`), we should get a new shape $\forall \mathtt{Y}. \{\mathtt{X} \mapsto \mathtt{p}[\mathtt{A} = \langle \mathtt{Y} \rangle] : \mathtt{A}\}$.

Lemma 3 (Thinning and renaming preserves well-formedness)**.** If $\tilde{\Xi}$ is well-formed, for all total r_R and partial r_P, $\mathsf{rnthin}(r_R; \tilde{\Xi}; r_P)$ is well-formed.

4.4 Component mixin linking

Given $\mathsf{link}()$ and $\mathsf{rnthin}()$, we are well equipped to say how to mixin link an entire resolved component (Figure 10). Our general strategy will be to compute the component shape of every $rdecl$ (as well as its elaboration to $mdecl$) and link them all together in the component judgment, combining all of the $mdecl$s together to form the final elaborated mixed component. There is an auxiliary definition (Definition 6) which computes the final renaming on the provisions of the component, so that only `exposed-modules` and `reexported-modules` are exposed to the world.

The rules for shaping modules and signatures are straightforward: a module m provides the module identity $P{:}m$, where P is the component identity of the component currently being shaped; a signature m adds a requirement $\langle m \rangle$. The rule for includes is more interesting: we look up the shape of the included unit from the context ($\tilde{\Gamma}$), and then rename it according to the user provided renamings rns. There is an auxiliary definition (Definition 5) for translating renaming syntax into a pair of module renamings.

Our presentation of the judgment is slightly nondeterministic in that it assumes we know the set of requirements Θ upfront to compute P; however it's easy to see that P is only used for the module identities assigned to module declarations, and thus does not actually affect the final requirements of a component.

4.5 Auxiliary definitions

Definition 5 (Interpretation of thinning and renaming)**.**

$$\mathsf{getrns}(\forall\Theta.\,\tilde{\Sigma}, [(\overline{rn_P})]\ [\texttt{requires}\ (\overline{rn_R})]) \;=\; (r_P, r_R)$$

The syntactic `include` declaration specifies renamings for provisions and requirements, but users are allowed to omit a renaming altogether (in which case everything is brought into scope) or give only a partial requirement renaming (in which case the requirement

$$\boxed{\tilde{\Gamma} \vdash rcomp \rightsquigarrow mcomp \triangleright \tilde{\Xi}}$$

$$\frac{\begin{array}{c} P = p[\Theta] \qquad \forall i:\ \tilde{\Gamma}; P \vdash rdecl_i \rightsquigarrow mdecl_i \triangleright \tilde{\Xi}_i \\ (\tilde{\Xi}, S) = \mathsf{link}(\overline{\tilde{\Xi}_i}^{\,i}) \qquad \forall\Theta.\,\tilde{\Sigma} = \tilde{\Xi} \qquad r_P = \overline{\mathsf{provs}(rdecl_i)}^{\,i} \end{array}}{\begin{array}{rcl} \tilde{\Gamma} \vdash \texttt{component}\ p\ \{\overline{rdecl_i}^{\,i}\texttt{\}} & \rightsquigarrow & \mathsf{component}\ p\ \Theta\ \{\overline{mdecl_i[\![S]\!]}^{\,i}\} \\ & \triangleright & \mathsf{rnthin}(\,\cdot\,; \tilde{\Xi}; r_P) \end{array}}$$

$$\boxed{\tilde{\Gamma}; P \vdash rdecl \rightsquigarrow mdecl \triangleright \tilde{\Xi}}$$

$$\frac{\begin{array}{c} p \triangleright \tilde{\Xi} \in \Delta \qquad \tilde{\Xi} = \forall\Theta.\,\tilde{\Sigma} \\ (r_R, r_P) = \mathsf{getrns}(\tilde{\Xi}, rns) \qquad \tilde{\Xi}' = \mathsf{rnthin}(r_R; \tilde{\Xi}; r_P) \end{array}}{\begin{array}{l} \tilde{\Gamma}; P \vdash\ \texttt{backpack-include:}\ p\ rns \rightsquigarrow \\ \qquad \mathsf{dependency}\ p[r_R]\ (r_P) \triangleright \tilde{\Xi}' \end{array}}$$

$$
\begin{array}{ll}
\tilde{\Gamma}; P \vdash & \texttt{exposed-module:}\ m\ \{hsbody\} \rightsquigarrow \\
& \mathsf{module}\ m\ \{hsbody\} \quad \triangleright \quad \{m \mapsto P{:}m\} \\
\tilde{\Gamma}; P \vdash & \texttt{other-module:}\ m\ \{hsbody\} \rightsquigarrow \\
& \mathsf{module}\ m\ \{hsbody\} \quad \triangleright \quad \{m \mapsto P{:}m\} \\
\tilde{\Gamma}; P \vdash & \texttt{required-signature:}\ m\ \{hssig\} \rightsquigarrow \\
& \mathsf{signature}\ m\ \{hssig\} \quad \triangleright \quad \forall \langle m \rangle.\ \{\} \\
\tilde{\Gamma}; P \vdash & \texttt{reexported-module:}\ m\ \texttt{as}\ m' \rightsquigarrow \\
& \varnothing \quad \triangleright \quad \{\}
\end{array}
$$

Figure 10. Component mixin linking. We implicitly lift $\Theta = \overline{\langle m \rangle}$ into a substitution $\overline{m = \langle m \rangle}$.

renaming is extended to be total over Θ.) getrns simply computes the appropriate r_R and r_P given this syntax.

Definition 6 (Interpretation of exposed/reexported modules)**.**

$$
\begin{array}{lcl}
\mathsf{provs}(\texttt{exposed-module:}\ m\ \{\_\}) & = & m \mapsto m \\
\mathsf{provs}(\texttt{reexported-module:}\ m\ \texttt{as}\ m') & = & m \mapsto m' \\
\mathsf{provs}(\_) & = & \cdot
\end{array}
$$

Only modules specified with `exposed-module` or `reexported-module` should be put in the final shape of a component. provs computes the appropriate r_P based on these declarations.

5. Type checking

We now give a formal model for how our implementation type-checks mixed components which still have unfilled requirements. It's not possible to give a full semantics, since that would involve formalizing all of Haskell as implemented by GHC, but we will

informally explain the operation of the judgments we assume are given to us.

5.1 Preliminaries

The context There is a bit of accounting to do:

- The component type context $\Gamma ::= \overline{p : \Xi}$ records the types of all components we have previously typechecked; these components are typed but not instantiated. Type lookup will find a component type and the instantiate it to get an actual type.
- The module variable context $\Theta ::= \overline{\langle m \rangle}$ specifies the set of module variables which are in scope. This context is computed by mixin linking and recorded in the Θ in $mcomp$.
- The name variable context $\theta ::= \overline{\{m.n\}}$ specifies the set of name variables that are in scope. Each should be ascribed a type in Σ_R (described next.)

$$\boxed{\Gamma; \Theta; \theta; \Delta; \mathcal{L}; p_0; m \vdash hsbody : \tau}$$

$$\boxed{\Gamma; \Theta; \theta; \Delta; \mathcal{L}; p_0; m \vdash hssig}$$

$$\boxed{ty <: ty'}$$

Figure 11. Assumed Haskell judgments.

- The local requirements context $\Sigma_R ::= \overline{m : \tau}$ specifies the module types of the requirements of the current component. In this formal development, we assume that Σ_R is given to us nondeterministically, see Section 7.2 for more details.
- The local provisions context $\Sigma_P ::= \overline{m : \tau}$ specifies the module types of modules defined in the current component. Collectively, $\Delta ::= (\Sigma_R; \Sigma_P)$ is the full local context, which will be transformed into the final type of the overall component.
- The import context $\mathcal{L} ::= \overline{m \mapsto M}$ maps module names to module identities, and specifies which module is actually imported by an `import` declaration in Haskell source, bringing the exports of that module into scope.
- The current component identifier p_0 says what the current component is, and is used to generate local module identities and say when a type is local as opposed to global.

Definition 7 (Name substitutions)**.** Intuitively, name substitutions refine name variables to more specific original names; in Section 3.4, this operation was shown diagrammatically by drawing a line between two entities. Formally, a name substitution $Sn ::= \overline{m.n = N}$ can be applied to a semantic object x with notation $x[\![Sn]\!]$ (the same notation as module substitution). The application is also functorial; here are the important cases:

$$\begin{array}{rcl} \{m.n\}[\![\cdot]\!] & =_N & \{m.n\} \\ \{m.n\}[\![m.n = N, Sn]\!] & =_N & N \end{array}$$

$$
\begin{array}{rcl}
\{m.n\}[\![m'.n' = N, Sn]\!] & =_N & \{m.n\}[\![Sn]\!] \quad (m.n \neq m'.n') \\
(M.n)[\![Sn]\!] & =_N & M.n \\
\overline{N}[\![Sn]\!] & =_{Ns} & \overline{N[\![Sn]\!]} \\
\{\overline{m : \tau}\}\,[\![Sn]\!] & =_\Sigma & \{\overline{m : \tau[\![Sn]\!]}\} \\
(\mathsf{iface}\ (Ns)\ \{\overline{ty}\})[\![Sn]\!] & =_\tau & \mathsf{iface}\ (Ns[\![Sn]\!])\ \{\overline{ty[\![Sn]\!]}\}
\end{array}
$$

5.2 Assumed Haskell judgments

In Figure 11, we give the black box judgments which we assume the compiler provides for us. The module typechecking judgment synthesizes a module type for the given $hsbody$. Each entity n freshly defined in this module is given the original name $p_0[\Theta]{:}m.n$, which will appear in the module type τ. For example, the type of module `Y`, defined in the component `p` from 8(b), exports the freshly named $\mathtt{p}[\mathtt{A} = \langle \mathtt{A} \rangle, \mathtt{B} = \langle \mathtt{B} \rangle]{:}\mathtt{Y.L}$.

The signature judgment does not synthesize a type for a signature $hssig$, in contrast to the module typechecking judgment. Instead it merely checks that the given $hssig$ is *consistent* with the requirement types provided by the local context Δ, which were nondeterministically chosen. In a type system where requirement types from dependencies automatically merge together, there is no reason to expect the syntactic $hssig$ specifies the full requirement for all the dependencies of the component (the user may have omitted it for it to be inferred.)

Finally, the subtyping judgment specifies whether or not a defined entity specification is a subtype of another specification. Generally, this is simply equality, except that any type is a subtype of a same-kinded abstract type (`data A` with no constructors.)

$\boxed{\Gamma; \Theta; \theta; \Delta; p_0 \vdash P : \Sigma_P}$

$$\frac{p \neq p_0 \qquad p : \forall \Theta_R.\, \forall \theta_R.\, \Sigma_R \to \Sigma_P \in \Gamma \qquad \Gamma; \Theta; \theta; \Delta; p_0 \vdash S : \forall \Theta_R.\, \forall \theta_R.\, \Sigma_R \Rightarrow Sn}{\Gamma; \Theta; \theta; \Delta; p_0 \vdash p[S] : \Sigma_P[\![S]\!][\![Sn]\!]}$$

$$\Gamma; \Theta; \theta; (\Sigma_R; \Sigma_P); p_0 \vdash p_0[\overline{m_i = \langle m_i \rangle}] : \Sigma_P$$

$\boxed{\Gamma; \Theta; \theta; \Delta; p_0 \vdash M : \tau}$

$$\frac{\Gamma; \Theta; \theta; \Delta; p_0 \vdash P : \Sigma_P \qquad m : \tau \in \Sigma_P}{\Gamma; \Theta; \theta; \Delta; p_0 \vdash P{:}m : \tau}$$

$$\frac{\langle m \rangle \in \Theta \qquad \Delta = \Sigma_R; \Sigma_P \qquad m : \tau \in \Sigma_R}{\Gamma; \Theta; \theta; \Delta; p_0 \vdash \langle m \rangle : \tau}$$

$\boxed{\Gamma; \Theta; \theta; \Delta; p_0 \vdash N :: ty}$

$$\frac{\Gamma; \Theta; \theta; \Delta; p_0 \vdash M : \tau \qquad n :: ty \in \tau}{\Gamma; \Theta; \theta; \Delta; p_0 \vdash M.n :: ty}$$

$$\frac{\{m.n\} \in \theta \qquad \Gamma; \Theta; \theta; \Delta; p_0 \vdash \langle m \rangle : \tau \qquad n :: ty \in \tau}{\Gamma; \Theta; \theta; \Delta; p_0 \vdash \{m.n\} :: ty}$$

Figure 12. Rules for type lookup

5.3 Type lookup and export/signature matching

We now define the mutually recursive judgments for *type lookup* (Figure 12) and *export/signature matching* (Figure 13), arguably the most important rules of our type system. Intuitively, computing the type of $p[S]$ simply involves taking the component type for p, wiring it up according to S, and then reading off the provided types under this wiring diagram. Formally, the wiring diagram represents a name substitution Sn which refines the name variables in p's type; we must compute this substitution by export/signature matching. The export matching rule, in turn, requires us to compute the types of the module identities in S (we are traversing the whole wiring diagram). Assuming that module identities are finite, this process is guaranteed to terminate.

The rules for type lookup are fairly straightforward. There are special cases for looking up (1) the component type of the local component p_0, (2) the module type of a module variable $\langle m \rangle$, and (3) the Haskell type of a name variable $\{m.n\}$.

The rule for export and signature matching is slightly more intricate. The export matching rule takes as input both the module substitution being applied S, as well as the component requirement Σ_R that the substitution is being applied to. We first must compute the types for every module in the substitution (the recursive process), and then compute an appropriate name substitution Sn induced by the entire substitution which unifies the export lists (intuitively, we just look at the export lists and unify the ones which have the same n). Finally, we perform signature matching, which simply checks that required type τ, after name substitution, is consistent with all the types in the context. It does this by going through each export of the requirement which is locally defined and checking the local

type.

5.4 Typechecking a mixed component

The rest of the typechecking process (Figure 14) involves processing each declaration of a mixed component in order, modifying the local contexts as we process the each declaration of the mixed com-

$$\boxed{\Gamma;\Theta;\theta;\Delta;p_0 \vdash S : \forall\Theta_R.\,\forall\theta_R.\,\Sigma_R \Rightarrow Sn}$$

$$\frac{\begin{array}{ll} S &= (\overline{m_i = M_i}^{\,i}) \\ \Theta_R &= \{\overline{m_i}^{\,i}\} \\ \theta_R &= \mathsf{dom}(Sn) \end{array} \qquad \forall i: \begin{cases} \Gamma;\Theta;\theta;\Delta;p_0 \vdash M_i : \tau_i' \\ \mathsf{exps}(\tau_i') \supseteq \mathsf{exps}(\tau_i[\![S]\!][\![Sn]\!]) \\ \Gamma;\Theta;\theta;\Delta;p_0 \vdash \tau_i[\![S]\!][\![Sn]\!]\ \mathsf{valid} \end{cases}}{\Gamma;\Theta;\theta;\Delta;p_0 \vdash S : \forall\Theta_R.\,\forall\theta_R.\,(\overline{m_i : \tau_i}^{\,i}) \Rightarrow Sn}$$

$$\boxed{\Gamma;\Theta;\theta;\Delta;p_0 \vdash \tau\ \mathsf{valid}}$$

$$\frac{\forall i: \quad \Gamma;\Theta;\theta;\Delta;p_0 \vdash M_i.n_i :: ty_i' \quad \wedge \quad ty_i' <: ty_i}{\Gamma;\Theta;\theta;\Delta;p_0 \vdash \mathsf{iface}\ (\overline{M_i.n_i}, \overline{N_j})\ \{\overline{n_i :: ty_i}\}\ \mathsf{valid}}$$

Figure 13. Rules for export matching and signature matching. $\mathsf{exps}(\tau)$ refers to the the export specification of the type τ.

ponent. Most per-declaration rules are fairly straightforward: mod-

ule typechecking produces a τ which is added to Σ_P and signature typechecking just checks for consistency with Σ_R.

However, the dependency rule is a little subtle. First, we eagerly typecheck $p[S]$ to ensure that it indeed has a type; not because we need to use the type, but because there is no guarantee that $p[S]$ is a well-typed instantiation; we need to check. Second, we need to consult the shape context in order to determine what set of modules (substituted according to S) we actually bring into scope for import.

Finally, the overall rule for $mcomp$ nondeterministically synthesizes the local requirement context Σ_R and then runs successively typechecks each declaration, starting with the empty local contexts. The results are all bundled up into the final component type of a component. The synthesized requirement has the following properties: (1) it is specific enough to permit the component to typecheck, (2) it doesn't contain unnecessary exports or type declarations, and (3) it is as abstract as possible with respect to original names (the more name variables, the better.)

6. Evaluation

Backpack is implemented as a set of patches to GHC and Cabal. We've tested our implementation in a number of ways:

An alternate package language A benefit of having a separation between mixin linking and typechecking is that it is a simple matter to swap the frontend language with something else. To assist in testing Backpack, we implemented a simple alternate frontend, based off of Backpack'14, which typechecks component definitions of the form:

```
component p where
    include base
    signature H where
        data T
    module M where
        import H
```

Such inlined module and signature definitions are extremely handy for test-cases, although they are not so helpful for libraries with large modules.

Replacing build-depends with signatures To show that Backpack'16 supports real-world "modularity in the large," we took a few packages in the public Hackage repository and rewrote them to deprecate some of their `build-depends` in favor of signatures instead:

$$\boxed{\tilde{\Gamma}; \Gamma; \Theta; p_0; \theta; \Sigma_R \vdash \{\Sigma_P; \mathcal{L}\} \quad mdecl \quad \{\Sigma'_P; \mathcal{L}'\}}$$

$$\frac{\Gamma; \Theta; \theta; (\Sigma_R; \Sigma_P); \mathcal{L}; p_0; m \vdash hsbody : \tau}{\tilde{\Gamma}; \Gamma; \Theta; p_0; \theta; \Sigma_R \vdash \left\{ \begin{matrix} \Sigma_P \\ \mathcal{L} \end{matrix} \right\} \mathsf{module}\ m\ \{hsbody\} \left\{ \begin{matrix} \Sigma_P, m : \tau \\ \mathcal{L} \end{matrix} \right\}}$$

$$\Gamma; \Theta; \theta; (\Sigma_R; \Sigma_P); \mathcal{L}; p_0; m \vdash hssig$$

$$\tilde{\Gamma}; \Gamma; \Theta; p_0; \theta; \Sigma_R \vdash \left\{ \begin{matrix} \Sigma_P \\ \mathcal{L} \end{matrix} \right\} \ \mathsf{signature}\ m\ \{hssig\}\ \left\{ \begin{matrix} \Sigma_P \\ \mathcal{L} \end{matrix} \right\}$$

$$\frac{\begin{matrix} p \triangleright \forall \Theta'.\, \tilde{\Sigma} \in \tilde{\Gamma} \qquad \Gamma; \Theta; \theta; \Delta; p_0 \vdash p[S] : \Sigma \\ r = \overline{m \mapsto m'} \qquad \overline{M} = \tilde{\Sigma}(m')[\![S]\!] \end{matrix}}{\tilde{\Gamma}; \Gamma; \Theta; p_0; \theta; \Sigma_R \vdash \left\{ \begin{matrix} \Sigma_P \\ \mathcal{L} \end{matrix} \right\} \mathsf{dependency}\ p[S]\ (r) \left\{ \begin{matrix} \Sigma_P \\ \mathcal{L}, \overline{m \mapsto M} \end{matrix} \right\}}$$

$$\boxed{\tilde{\Gamma}; \Gamma; \Theta; p_0; \theta; \Sigma_R \vdash \{\Sigma_P; \mathcal{L}\} \quad \overline{mdecl} \quad \{\Sigma'_P; \mathcal{L}'\}}$$

$$\frac{}{\tilde{\Gamma}; \Gamma; \Theta; p_0; \theta; \Sigma_R \vdash \{\Sigma_P; \mathcal{L}\} \quad \cdot \quad \{\Sigma_P; \mathcal{L}\}}$$

$$\frac{\begin{matrix} \tilde{\Gamma}; \Gamma; \Theta; p_0; \theta; \Sigma_R \vdash \{\Sigma_P; \mathcal{L}\} \quad mdecl \quad \{\Sigma'_P; \mathcal{L}'\} \\ \tilde{\Gamma}; \Gamma; \Theta; p_0; \theta; \Sigma_R \vdash \{\Sigma'_P; \mathcal{L}'\} \quad \overline{mdecl'} \quad \{\Sigma''_P; \mathcal{L}''\} \end{matrix}}{\tilde{\Gamma}; \Gamma; \Theta; p_0; \theta; \Sigma_R \vdash \{\Sigma_P; \mathcal{L}\} \quad mdecl, \overline{mdecl'} \quad \{\Sigma''_P; \mathcal{L}''\}}$$

$$\boxed{\tilde{\Gamma}; \Gamma \vdash mcomp : \Xi}$$

$$\frac{\begin{matrix} (\forall \theta.\, \Sigma_R)\ \text{most general s.t.} \\ \begin{cases} \Theta = \mathsf{dom}(\Sigma_R)\ \wedge\ \theta = \mathsf{fnv}(\Sigma_R)\ \wedge\ \forall \{m.n\} \in \theta : \langle m \rangle \in \Theta \\ \tilde{\Gamma}; \Gamma; \Theta; p; \theta; \Sigma_R \vdash \{\cdot; \cdot\}\ \overline{mdecl}\ \{\Sigma_P; \mathcal{L}\} \end{cases} \end{matrix}}{\tilde{\Gamma}; \Gamma \vdash \mathsf{component}\ p\ \Theta\ \{\overline{mdecl}\} : \forall \Theta.\, \forall \theta.\, \{\Sigma_R\} \to \{\Sigma_P\}}$$

$$\boxed{\square \ \leq \ \square \quad \text{less general than}}$$

$$\forall\theta_1.\Sigma_1 \;\leq\; \forall\theta_2.\Sigma_2 \quad\Leftrightarrow\quad \exists Sn : \begin{cases} \mathsf{dom}(Sn) = \theta_2 \\ \Sigma_1 \;\leq\; \Sigma_2[\![Sn]\!] \end{cases}$$

$$\begin{array}{rcl} (\overline{m' : \tau'}, \overline{m : \tau_1}) & \leq & (\overline{m : \tau_2}) \quad \Leftrightarrow\; \overline{\tau_1 \leq \tau_2} \\ \mathsf{iface}\ (Ns', Ns)\ \{\overline{ty'}, \overline{ty_1}\} & \leq & \mathsf{iface}\ (Ns)\ \{\overline{ty_2}\} \;\Leftrightarrow\; \overline{ty_1 <: ty_2} \end{array}$$

Figure 14. Rules for mixed component language.

- We rewrote `binary-0.8.0.0`[7] to have signatures for `bytestring`, `containers` and `array`, demonstrating that GHC's core libraries can be modularized over. (23 signatures)
- We rewrote `ghc-simple-0.3`[8] to have signatures for `ghc`, demonstrating that the GHC API (a very complicated API) can be modularized over. (57 signatures)

For example, here is the signature we wrote for `Data.Array.IArray` in `binary`, exercising many of Haskell's features including type classes:

```
{-# LANGUAGE ... #-}
module Data.Array.IArray(
  module Data.Array.IArray,
  module Data.Ix
```

[7] `https://hackage.haskell.org/package/binary-0.8.0.0`

[8] `https://hackage.haskell.org/package/ghc-simple-0.3`

```
  ) where

import Data.Ix

type role Array nominal representational
class IArray (a :: * -> * -> *) e
data Array i e
instance IArray Array e
bounds :: (IArray a e) => forall i. Ix i
                       => a i e -> (i, i)
```

Adding signatures to packages was a straightforward (if a little tedious) process. One thing we discovered, however, was that the lack of *recursively dependent signatures* [3] (*i.e.*, signatures that form an import cycle) sometimes caused problems when a dependency was implemented using `hs-boot` files to implement recursion. However, this could be easily worked around by adding a "synthetic" signature which collected all of the mutually dependent data types together, and then have the real signatures reexport from this signature; the synthetic signature is then implemented using a dummy module. Here is an example of such a synthetic signature from `ghc-simple`'s signatures for GHC:

```
module RecTypes where

import ConLike  (ConLike)
import CoAxiom  (Branched, CoAxiom)
import OccName  (OccName)
```

```
data Id      -- from Id,   imports Name
data Name    -- from Name, imports Type
data TyCon   -- from Type
data TyThing -- from Type, imports Id
    = AnId Id
    | AConLike ConLike
    | ATyCon TyCon
    | ACoAxiom (CoAxiom Branched)
```

Without this synthetic signature, the signatures for `Id`, `Name`, and `Type` would form a cycle.

7. Limitations

Some significant tasks remain to complete Backpack'16. We discuss some of the limitations of the current system here.

7.1 Metatheory

We have not rigorously done proofs on Backpack'16's metatheory; as such, we can only conjecture that in the fragment of Haskell without type classes and type families, successful separate typechecking of components implies successful linking as well.

One thing that is troublesome about the theorem in Backpack'14, however, is that it simply is not true for full Haskell. Specifically, open type families [15] are inherently non-modular, as they introduce axioms to Haskell which cannot be hidden by signatures. Any module system which supports such open type families must either (1) impose a strict *orphan* constraint, so that declared axioms in separate modules are guaranteed not to con-

flict, or (2) allow for the possibility that linking can fail, even if the components typechecked separately.

There is a soundness result we can report, however: the soundness of *compiled* Backpack'16 code reduces to the soundness of GHC Haskell. This is because we retypecheck and compile every instantiation of a component, and thus the compilation process reduces to ordinary Haskell compilation. The retypechecking step during compilation is where things like incompatible open type

family axioms are found. Indeed, it would be possible to skip the typechecking step described in this paper altogether.

7.2 Signature merging

In our tour of Backpack, we stated that requirements from multiple components automatically merge together when they are brought into scope. Indeed, we have implemented this; unfortunately, it is unclear how to formally specify this process without substantially complicating the typechecking rules, which is why our typechecking semantics currently nondeterministically guess the "correct" requirement type. The problem is that in our semantics, when we do component instantiation, it is assumed that you know the module types of all of the inputs to the component, including the module types of local requirements. However, to determine a type of a requirement to merge in, you need to be able to instantiate a component prior to knowing how all of its holes are instantiated.

We have not yet found a satisfactory way of formalizing this algorithmically. Of course, using the semantics for Backpack'14, the merged type can be determined; however, to implement Backpack'14 directly would require successively refining the types of

modules we have already typechecked, which is a poor match for a one source file, one output file compilation model. It is also possible to assume that the locally written signature is complete (this is implemented as a flag) and not attempt to do any merging. It would be simple to adjust the rules for this case.

8. Related Work

Comparison with Backpack'14 The inspiration for this work was the original Backpack paper [7]. Backpack was first to pose the problem of retrofitting Haskell with interfaces, and many of its design ideas, such as mixin packages, applicativity and module identities have been preserved in this paper. The contribution of this paper is an actual *implementation* of the Backpack design, by refactoring of these ideas into a form that can be implemented in two stages: mixin linking handled by the package manager, and typechecking and compilation handled by the compiler.

This refactoring necessitated a change to one of the core definitions in the package language: in Backpack'14, type equality was based on a per-module computed *module identity*. In effect, every defined module separately kept track of the set of signatures that it transitively imported. If mixin linking is not allowed to inspect source code, the notion of identity must be coarsened. In this paper, *instantiated component identities* track all of the requirements in the component, which are computed during mixin linking.

A technical accomplishment of the original Backpack was a solution to the "double vision problem", where the typechecker can see two distinct names for the same underlying type in a mutually recursive module. In Backpack, this problem was solved by way of a "shaping pass" which first computed the identities of what we call name variables prior to type checking. In this paper, we do not permit mutual recursion; thus, the double vision problem does not

occur, and our "shaping pass" only computes a very coarse-grained shape on the overall wiring structure of components. From this wiring structure, Backpack'16 can compute the correct identities for name variables at the same time as type checking.

We have also taken a different approach to defining the semantics of Backpack'16. Instead of elaborating to Haskell, which is not how you would ever implement a Backpack-like system, we give an ad hoc semantics which directly reflects how Backpack'16 has been implemented in GHC.

Modularity in the package manager While there is far more literature on module systems that can be implemented entirely by a compiler, there has been some work which has looked at the problem of modular development at the package level.

One such system is the Functoria DSL[9] of MirageOS [11]. MirageOS is a library operating system written in OCaml, which provides modules and functors for constructing unikernels. Rather than manually instaitate these functors, users write in the Functoria DSL, which describes what dependencies to install (via the OCaml package manager) and how the ML functors should be assembled. Unlike Backpack'16, their DSL follows the model of explicit functor applications rather than mixin linking.

The Nix package manager [5] is a system for enabling reproducible builds of packages. Nix defines a (pure, functional) language of component *derivations—i.e.*, the source code and configuration needed to build the derived component—functorized over configuration parameters and derivations of depended-upon components. Components are linked together with explicit functor ap-

plication, albeit with some of the syntactic convenience of mixin linking. However, there is no type system for components, and thus the Nix output hashes (similar to our component identifiers) only serve the role of uniquely identifying derivations.

The SMLSC extension to Standard ML [16], while primarily intended as a mechanism to support separate compilation in Standard ML, also has some similarities to Backpack'16. Like Backpack, SMLSC operates at the level of *units* (our components), and defines interfaces between units to allow them to be separately typechecked. Unlike Backpack'16, SMLSC does not support reusing units with different implementations of their interfaces: dependencies in SMLSC are always *definite references*, and signatures are used purely to permit separate compilation. In SMLSC, if you want multiple instantiations, you are expected to use ML functors.

An unusual case of *not* using a package manager when it would be useful occurs in C++ templates.[10] C++ templates are applicative, in the sense that two occurrences of `vector<int>` refer to the same type. However, the C++ compiler must generate code when a template is instantiated. Implemented naively, this could result in a lot of duplicate copies of code. One early method of handling this problem, "Cfront model", involved a template database where instances of templates were maintained. However, it was too complicated for most C++ compilers to handle this database, and so the usual "Borland model" (implemented by GCC, among others) is to just recompile every template instantiation and deduplicate them at link time. With Backpack, we already have a package manager, Cabal, which administers its own installed package database, so we can offload the caching of instantiated components to it. (This technique would not work for C++ templates, whose type based dispatch must be deeply integrated with the compiler.)

ML functors The original Backpack language distinguished itself from "functors" in (variants of) the ML module system [8, 12] by the fact that it supported separate type checking for recursive modules under applicative instantiation. Additionally, by being a mixin system, it is a better fit for the package language and avoids the need for sharing constraints.

As this paper does not address mutual recursion, one may wonder if the mixed component language is not simply just a stylized applicative functor language. In fact, it is! The reason our technical presentation is done in the way it is done here is because our primary goal was integrating with the existing compiler infrastructure.

Mixin linking There is a rich literature in the mixin linking world, which both Backpack and Backpack'16 draw heavily from [1, 6, 13]. Indeed, the relationship to this literature is even clearer in Backpack'16, as the mixin linking step is factored out and is independent of the Haskell language. For example, the basic algorithm for linking in Cardelli's *linksets* calculus [2], at a high level,

[9] `https://mirage.io/blog/introducing-functoria`

[10] `https://gcc.gnu.org/onlinedocs/gcc/Template-Instantiation.html`

is essentially the same algorithm as our mixin linking. The difference, however, is that we must keep track of the structure of the "wiring diagram", as this structure will be used to establish the identities of types at the Haskell level. In contrast, Cardelli gave no account of the interaction between module-level linking and core-level user-defined abstract data types.

The object oriented community has also studied mixin-style composition in their designs. However, these mechanisms are organized around dynamic binding and objects; whereas in Backpack-style systems, the emphasis is on packages. Users of Backpack'16

pay no performance penalty switching from a direct dependency to an indirect dependency via a signature, because we don't do separate compilation of Backpack'16.

References

[1] Davide Ancona and Elena Zucca. A calculus of module systems. *JFP*, 12(2), 2002.

[2] Luca Cardelli. Program fragments, linking, and modularization. In *POPL '97*.

[3] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI '99*.

[4] Edsko de Vries. Qualified goals in the Cabal solver. Technical report, Well Typed, 2015.

[5] Eelco Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Utrecht University, 2006.

[6] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *PLDI '98*.

[7] Scott Kilpatrick, Derek Dreyer, Simon Peyton Jones, and Simon Marlow. Backpack: Retrofitting Haskell with interfaces. In *POPL '14*.

[8] Xavier Leroy. The Objective Caml system: Documentation and user's manual.

[9] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[10] Andres Loh and Duncan Coutts. A new modular dependency solver for cabal-install. Technical report, Well Typed, 2011.

[11] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *ASPLOS '13*.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[13] Andreas Rossberg and Derek Dreyer. Mixin' up the ML module system. *ACM TOPLAS*, 35(1), 2013.

[14] Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. *JFP*, 24(5), 2014.

[15] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *ICFP '08*, 2008.

[16] David Swasey, Tom Murphy VII, Karl Crary, and Robert Harper. A separate compilation extension to Standard ML. In *ML '06*.