Backpack: Retrofitting

Scott Kilpatrick
MPI-SWS
skilpat@mpi-sws.org

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Abstract

Module systems like that of Haskell permit only a weak form of modularity in which module implementations depend directly on other implementations and must be processed in dependency order. Module systems like that of ML, on the other hand, permit a stronger form of modularity in which explicit interfaces express assumptions about dependencies, and each module can be typechecked and reasoned about independently.

In this paper, we present Backpack, a new language for building separately-typecheckable *packages* on top of a weak module system like Haskell's. The design of Backpack is inspired by the MixML module calculus of Rossberg and Dreyer, but differs significantly in detail. Like MixML, Backpack supports explicit interfaces and recursive linking. Unlike MixML, Backpack supports a more flexible *applicative* semantics of instantiation. Moreover, its design is motivated less by foundational concerns and more

by the *practical* concern of integration into Haskell, which has led us to advocate simplicity—in both the syntax and semantics of Backpack—over raw expressive power. The semantics of Backpack packages is defined by elaboration to sets of Haskell modules and binary interface files, thus showing how Backpack maintains interoperability with Haskell while extending it with separate typechecking. Lastly, although Backpack is geared toward integration into Haskell, its design and semantics are largely agnostic with respect to the details of the underlying core language.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Recursion, Abstract data types, Modules; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

Keywords Type systems; mixin modules; Haskell modules; applicative instantiation; recursive modules; separate modular development; packages; module systems

1. Introduction

Suppose an author A wants to write, test, and publish a software component (or "package") P, that needs to call a random-number generator. But A wants each customer C to be able to supply his or her *own* random-number generator. In a typed language, the author A must define the *interface* to the random-number generator,

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01. . . $15.00.
http://dx.doi.org/10.1145/2535838.2535884

Haskell with Interfaces

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Simon Marlow
Facebook
marlowsd@gmail.com

typecheck P with respect to the interface, and publish P. The client C then links P with a particular random-number generator that matches the interface. We refer to this style of development as *separate modular development* (SMD), as distinct from the style of *incremental modular development* (IMD) in which a package can only be typechecked when the implementations of its dependencies are available.

One of the most prominent approaches to SMD is that taken by the ML module system and its many variants [22, 21]. ML provides *functors*, which enable a module M to be *parameterized* over the implementations of its dependencies; the dependencies can then be

instantiated by functor application in multiple different ways, even within a single program.

An alternative approach is to use *mixin modules* [4, 3, 13, 11]. Instead of relying on parameterization, mixin modules support SMD by combining within their namespaces both *defined* and *undefined* components, the latter specified via interfaces. Unlike functors, mixin modules support *recursive linking*; and since linking is implicitly "by-name", mixins also avoid the propagation of coherence (or "sharing") constraints so common with functor programming. Moreover, recent work by Rossberg and Dreyer on the MixML type system [27] has demonstrated that mixin modules have the capacity to *subsume* the functionality of ML modules.

However, despite their advantages, mixin modules have yet to be adopted by any widely-used statically-typed functional language.[1] In the case of the ML languages, this is understandable: ML already has a powerful module system, and the extra benefit afforded by mixins is arguably not worth the replacement cost.

What about Haskell? Haskell's existing module system was consciously designed as a weak namespace management system without a proper notion of interface [17, Section 8.2], and hence supports only IMD, not SMD. Tools like the Cabal package management system pick up the slack, enabling users to (ab)use version-range dependencies in order to work around the lack of interfaces. But the Haskell community recognizes that this is a makeshift solution and is actively seeking ways to support SMD properly.

In short, Haskell is a prime example of a language that is ripe for extension with interfaces and mixins. The trouble is that the foundational accounts of mixin modules that have appeared in the literature [3, 16, 27] employ a variety of complex and unconventional type systems, and it is not at all clear how to convert any of

them into a practical design that could be realistically incorporated into a language like Haskell.

With this in mind, we make the following contributions:

- We present Backpack, a new language for building *mixin-style packages* on top of an existing (weak) module system like Haskell's (Section 2). Like MixML [27], Backpack supports interfaces, recursive linking, abstract data types, and SMD.

[1] We are not counting Scala [24]: it supports mixin composition, but in a way that is integrally tied to its object-oriented mechanisms.

Package Names	P	$\in PkgNames$
Module Path Names	p	$\in ModPaths$
Package Respositories	R	$::= \overline{D}$
Package Definitions	D	$::= \textsf{package}\ P\ t\ \textsf{where}\ \overline{B}$
Bindings	B	$::= p = [M] \mid p :: [S] \mid p = p \mid \textsf{include}\ P\ t\ r$
Thinning Specs	t	$::= (\overline{p})$
Renaming Specs	r	$::= \langle \overline{p \mapsto p} \rangle$
Module Expressions	M	$::= \ldots$
Signature Expressions	S	$::= \ldots$

Figure 1. Backpack syntax.

- Unlike MixML, Backpack supports a more flexible *applicative* (rather than generative) semantics of instantiation [20], thereby extending mixin modules into new territory (Section 2.4). It would be easy to support generativity as well.
- Unlike other strong module systems, Backpack subordinates expressive power to *simplicity*, *practicality*, and *orthogonality* from the core language and its type system. The main technical device that supports this orthogonality is the central notion of *module identity* (Section 3.1), which we can treat largely independently from the type system of the core language. The type system for Backpack is *much* simpler than that of MixML.
- We give a formal description (Sections 3.2–3.4) of how to elaborate a Backpack package into a set of ordinary Haskell modules and module types (the latter corresponding to GHC's existing notion of "binary interface file"). If the package is complete (*i.e.*, fully implemented), it can be compiled and executed. But regardless, the Haskell modules output by elaboration can be *typechecked separately* from their missing dependencies.

- We prove soundness of Backpack's elaboration, which guarantees that a complete package will elaborate to a well-typed set of Haskell modules (Section 3.5). Even *stating* soundness required us to define a formal semantics for separate typechecking of (recursive) Haskell modules, which did not exist previously.

Finally, we conclude the paper in Sections 4 and 5 with a detailed discussion of related and future work. For space reasons, we leave a number of formal details to our accompanying technical appendix, available at `http://plv.mpi-sws.org/backpack/`.

2. A Backpack Tour

Figure 1 gives the syntax of Backpack. A *package definition* D gives a package name P to a a sequence of *bindings* $\overline{B}$. The simplest form of binding is a *concrete module binding*, of the form $p = [M]$, which binds the module name p to the implementation $[M]$. For example:

```
package ab-1 where
  A = [x = True]
  B = [import A; y = not x]
```

The code in square brackets represents module *implementations*, whose syntax is just that of a Haskell module (details in the technical appendix; Appendix §5). Indeed, in a practical implementation of Backpack, the term $[M]$ might be realized as the name of a file containing the module's code. However, note that the module lacks a header "module M where ..." because the module's name is given by the Backpack description.[2]

Package ab-1 binds two modules named A and B. The first module, bound to A, imports nothing and defines a core value x, and the second module, bound to B, imports the first module and

makes use of that `x` in its definition of `y`. The type of this package

[2] We still provide syntax for optional "export lists" of core language entities; only the module name disappears.

expresses that it contains a module A which defines `x :: Bool` and a module B which defines `y :: Bool`. (We will more precisely discuss types, at the package and module levels, in Section 3.)

The module bindings in a package are explicitly sequenced: each module can refer only to the modules bound earlier in the sequence. In fact the bindings should be interpreted as iteratively building up a local module context that tracks the name and type of each module encountered. For example, if the order of the two bindings were reversed, then this package would cease to be well-typed, as the module reference A would no longer make sense.

Module bindings do not shadow. Rather, if the same module name is bound twice, the two bindings are *linked*; see Section 2.3.

2.1 Top Level and Dependencies

A *package repository* consists of an ordered list of package definitions. Each package in a repository sees only those packages whose definitions occur earlier in the sequence. To make use of those earlier packages — *i.e.*, to depend on them — a package *includes* them using the **include** binding form, thus:

package abcd-1 **where**

```
C = [x = False]
include ab-1
D = [import qualified A
     import qualified C
     z = A.x && C.x    ]
```

One should think of an **include** construct as picking up a package and dumping all of its contents into the current namespace. In this case, the modules A and B are inserted into the package abcd-1 as if they were bound between C and D. Consequently the module bound to D can import both A and C. The type of abcd-1 says that it provides four modules: C (which provides `x :: Bool`), D (which provides `z :: Bool`), and the two modules A and B from package ab-1, even though they were defined there and merely included here. (The modules exposed by a package can be controlled with syntax that resembles that of the module level; this feature is discussed as a special case of *thinning* in Section 2.4.)

In this paper, we will treat the example package definitions as the bindings in a single package repository. At this point, that top level includes the definition for ab-1 followed by abcd-1.

2.2 Abstraction via Interfaces

Up to this point, the package system appears only to support IMD since each module can only be checked after those that it depends on. For example, abcd-1 could only be developed and checked *after* the package ab-1 had already been developed and checked; otherwise we would not be able to make sense of the import declaration import qualified A and the subsequent usage of `A.x` as a `Bool`.

To support SMD as well, Backpack packages may additionally contain *abstract module bindings*, or *holes*. To specify a hole, a developer provides a set of core-language declarations, called a *signature* S, and binds a module name p to it by writing $p :: [S]$. One should think of holes as obligations to eventually provide implementing modules; a package is not *complete* until all such obligations are met. Concrete modules, on the other hand, are simply those bound to actual implementations (as in all previous examples). This combination of abstract and concrete components reflects the mixin-module basis of our package system.

As our first example, we simulate how the `abcd-1` package might have been developed modularly by specifying holes for the "other" components, `A` and `B`:

```
package abcd-holes-1 where
  A :: [x :: Bool]
  B :: [y :: Bool]
  C = [... as before ...]
  D = [... as before ...]
```

By "stubbing out" the other components, the developer of abcd-holes-1 can typecheck her code (in C and D) entirely separately from the developer who provides A and B. In contrast, in the existing Cabal package system, developers cannot typecheck their package code without first choosing particular version instances of their dependencies. Effectively, they test the well-typedness of their code with respect to individual configurations of dependencies which may or may not be the ones their users have installed.

Manually writing the holes for depended-upon components, as above, involves too much duplication. Instead a developer can define a package full of holes that designates the interface of an entire component. A client developer includes that package of holes and thus brings them into her own package without writing all those signatures by hand. The following two packages achieve the same net result (and have the same type) as abcd-holes-1, but without signatures in the client package:

```
package ab-sigs where
  A :: [x :: Bool]
  B :: [y :: Bool]
```

```
package abcd-holes-2 where
  include ab-sigs
  C = [... as before ...]
  D = [... as before ...]
```

Holes are included in exactly the same manner as concrete modules, and they retain their status as holes after inclusion. Under the interpretation of holes as obligations, inclusion propagates the obligations into the including package.

In these two examples we have named the packages abcd-holes-1 and abcd-holes-2, which might suggest multiple versions of a single package abcd-holes (*e.g.*, in Cabal). However, while they may convey that informal intuition, in the present work we focus on modularity of packages, leaving a semantic account of versioning for future work.

2.3 Linking and Signature Matching

So far, all package examples have contained bindings with distinct names. What has appeared to be mere sequencing of bindings is actually a special case of a more general by-name linking mechanism: linking two mixin modules with strictly distinct names merely concatenates them. Whenever two bindings share the same name, however, the modules to which they are bound must themselves link together. This gives rise to **three cases**: hole-hole, mod-mod, and mod-hole.

First, when linking two holes together, we merge their two interfaces into one. This effectively joins together all the core language declarations from their respective signatures. The resulting hole provides exactly the entities that both original holes provided.

```
package yourlib where
  Prelude :: [data List a = ...]
  Yours   = [import Prelude; ...]

package mylib where
  Prelude :: [data Bool = ...]
  include yourlib
  Mine    = [import Prelude; import Yours; ...]
```

The mylib package above declares its own hole for Prelude and also includes the hole for Prelude from yourlib. Before the binding

for Mine is checked, the previous bindings of Prelude must have linked together. This module can see both `List` and `Bool` since they are both in the interface of the linked hole, whereas the Yours

module could only see the `List` datatype in Prelude. (Swapping the order of the first two bindings of mylib has no effect here.)

This example highlights another aspect of programming with mixin-based packages: each package has the option of writing *precise* interfaces for the other packages (*i.e.*, modules) it depends on. Specifically, yourlib only needs the `List` datatype from the standard library's Prelude module, rather than the entire module's myriad other entities. This results in a stronger type for yourlib since the assumptions it makes about the Prelude module are more precise and focused.

Not all interface merges are valid. For example, if mylib had also declared a `List` datatype, but of a different kind from that in yourlib (*e.g.*, `data List a b = ...`), then the merge would be invalid and the package would be ill-typed.

Second, when linking two module implementations together, it intuitively makes no sense to link together two *different* implementations since they define different code and different types. Rather than rejecting all mod-mod linkages, as for instance MixML [27] does, we instead insist that mod-mod linking only succeed if the two implementations are *the same*, in which case the linkage is a no-op. To test this, we require equivalence of their *module identities* (about which see Sections 2.4 and 3.1).

Consider the following classic diamond dependency:

```
package top where
  Top = [...]

package right where
  include top
  Right = [...]

package left where
  include top
  Left = [...]

package bottom where
  include left; include right
  Bottom = [...]
```

The bottom package of the diamond links together the packages left and right, each of which provides a module named Top that it got from the top package. The linking resulting from the inclusions in bottom is well-typed because left and right provide the *same* module Top from package top.

Third, when linking a module with a hole, the module's type must be a subtype of the hole's, and we say that the module "fills," "matches," or "implements" that hole. This form of linking most closely resembles the traditional concept of linking, or of functor application; it also corresponds to how structures match signatures in ML. Roughly, a module implements a hole if it defines all the entities declared in that hole and with the exact same specifications.

The mylib package above has a hole for the Prelude module. As this package is not yet complete, it can be typechecked, but not yet compiled and executed. (Supporting separate *compilation* would require sweeping changes to GHC's existing infrastructure.) We therefore link mylib with a particular implementation of its Prelude hole so that it may now be compiled and used:

```
package mylib-complete-1 where
  include mylib
```

$$\mathsf{Prelude} = \begin{bmatrix} \texttt{data List a = ...} \\ \texttt{data Bool = ...} \\ \texttt{null xs = ...} \end{bmatrix}$$

The implementation of Prelude provides the two entities declared in the hole (included from mylib) and an additional third entity, the value `null`. This implementation matches the interface of the hole, so the linkage is well-typed.

For simplicity, our definition of when a module matches a hole is based on *width* rather than *depth* subtyping. In other words, a module may provide more entities than specified by the hole it is filling, but the types of any values it provides must be the same as the types declared for those values in the hole's signature. In particular, the match will be invalid if the implemented types are more general than the declared types. For example, a polymorphic

```
package prelude-sig where
  Prelude :: [data List a = Nil | Cons a (List a)]

package arrays-sig where
  include prelude-sig
  Array :: [import Prelude
            data Arr (i::*) (e::*)
            something :: List (Arr i e)]

package structures where
  include arrays-sig
  Set    = [import Prelude; data S ...]
  Graph  = [import Prelude; import Array; data G ...]
  Tree   = [import Prelude; import Graph; data T ...]

package arrays-a where
  include prelude-sig
  Array = [import qualified Prelude as P
           data Arr i e = MkArr ...
           something = P.Nil]

package arrays-b where
  include prelude-sig
  Array = [import Prelude
           data Arr i e = ANil | ...
           something = Cons ANil Nil]

package graph-a (Graph, Prelude) where
  include arrays-a
  include structures (Graph, Prelude, Array)

package graph-b (Graph, Prelude) where
  include arrays-b
  include structures (Graph, Prelude, Array)

package multinst where
```

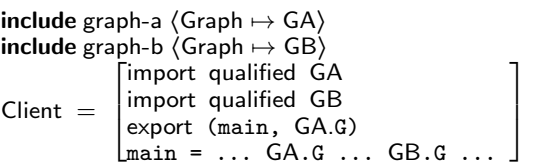

Figure 2. Running example: Data structures library and client.

identity function of type `forall a :: *. a -> a` will not match a hole that declares it as having type `Int -> Int`.

2.4 Instantiation and Reuse

Developers can reuse a package's concrete modules in different ways by including the package multiple times and linking it with distinct implementations for its holes; we call each such linkage an *instantiation* of the package. Furthermore, in Backpack, packages can be instantiated multiple times, and those distinct instantiations can even coexist in the same linked result. (In contrast, both Cabal and GHC currently prevent users from ever having two instantiations of a single package in the same program.)

Figure 2 provides an example of multiple instantiations in the multinst package, but this example employs a couple features of Backpack we must first introduce—*thinning* and *renaming*.[3]

[3] In our examples so far, we have omitted thinning specs entirely. But actually, according to Figure 1, all package definitions and inclusions should contain a thinning spec. To infer a thinning spec where one is omitted, one can simply list all module paths provided by the corresponding package. The syntax used in our examples can thus be translated into our formal

language with a straightforward (type-directed) rewriting. Moreover, the syntax for package inclusion given in Figure 1 requires that all inclusions additionally contain a renaming. When we omit renaming, it means that one should use the empty renaming, $\langle\rangle$.

The two packages arrays-a and arrays-b provide two distinct implementations of the Array module described by the hole specification in the earlier arrays-sig package. The next two packages grab the Graph implementation from structures and implement its Array hole with the respective array implementations. Since structures also defines Set and Tree, these (unwanted) modules would naively be included along with Prelude and Array and would thus pollute the namespaces of graph-a and graph-b. Instead, these packages *thin* the structures package upon inclusion so that only the desired modules, Prelude and Array, are added to graph-a and graph-b. (This closely resembles the *import* lists of Haskell modules, which may select specific entities to be imported.) Similarly, implementation details of a package definition can be hidden—rather than provided to clients—by thinning the definition to expose only certain module names. (This closely resembles the *export* lists of Haskell modules.) By thinning their definitions to expose only Prelude and Graph, both packages graph-a and graph-b hide the internal Array modules used to implement their Graph modules.

At this point, graph-a and graph-b provide distinct instantiations of the Graph module from structures, distinct in the sense that they do not have the same *module identity*. The identity of a module—a crucial notion in Backpack's semantics (see Sec-

tion 3.1)—essentially encodes a dependency graph of the module's source code. Since the Graph modules in graph-a and graph-b import two different module sources for the Array hole—one from arrays-a and the other from arrays-b—they do not share the same dependency graph and hence have distinct identities.

Thus, if the final package multinst were to naively include both graph-a and graph-b, Backpack would complain that multinst was trying to merge two distinct implementations with the same name. To avoid this error, the inclusions of graph-a and graph-b employ *renaming* clauses that rename Graph to GA and GB, respectively, so that the two Graph implementations do not clash.

One may wonder whether it is necessary to track dependency information in module identities: why not just generate fresh module identities to represent each instantiation of a package? To see the motivation for tracking more precise dependency information, consider the example in Figure 3. Both the applic-left and applic-right packages *separately* instantiate the Graph module from structures with the *same* Array implementation from arrays-a—*i.e.*, both instantiations refer to the same identity for Array. Backpack thus treats the two resulting Graph modules (and their G types) as one and the same, which means the code in applic-bot is well-typed. In other words, the identity of Graph inside applic-left is equivalent to that of Graph inside applic-right, and thus the G types mentioned in both packages are compatible.

As this example indicates, our treatment of identity instantiation exhibits sharing behavior. We call this an *applicative* semantics of identity instantiation, as opposed to a potential *generative* semantics in which the two instantiations—even when instantiated with the same identity—would produce distinct identities.

As is well known in the ML modules literature [20, 28], applica-

tivity enables significantly more flexibility regarding when module instantiation must occur in the hierarchy of dependencies. In the previous example, the authors of applic-left and applic-right were free to instantiate Graph *inside* their own packages. Under a generative semantics, on the other hand, in order to get the same Graph instantiation in both packages, it would need to be instantiated in an earlier package (like graph-a from Figure 2) and then included in both applic-left and applic-right; hence, the code as written in Figure 3 would under a generative semantics produce two distinct Graph identities and `G` types. As Rossberg *et al.* have noted [28], applicative semantics is generally safe only when used in conjunction with purely functional modules. It is thus ideally suited to Haskell, which isolates computational effects monadically.

```
package applic-left (Prelude, Left) where
  include structures
  include arrays-a
  Left = [import Graph; x :: G = ...]

package applic-right (Prelude, Right) where
  include arrays-a
  include structures
  Right = [import Graph; f :: G -> G = ...]

package applic-bot where
  include applic-left
  include applic-right
  Bot = [import Left; import Right; ... f x ...]
```

Figure 3. Example of applicativity.

2.5 Aliases

Occasionally one wants to link two holes whose names differ. The binding form $p = p$ in Figure 1 allows the programmer to add such aliases, which may be viewed as sharing constraints. For example:

```
package share where
  include foo1 (A, X)
  include foo2 (B, Y)
  X = Y
```

Here, A (from foo1) depends on hole X, and B (from foo2) on hole Y, and we want to require the two holes to be ultimately instantiated by the same module. The binding $\mathsf{X} = \mathsf{Y}$ expresses this constraint.

2.6 Recursive Modules

By using holes as "forward declarations" of implementations, pack-

ages can define recursive modules, *i.e.*, modules that transitively import themselves. The Haskell Language Report ostensibly allows recursive modules, but it leaves them almost entirely unspecified, letting Haskell implementations decide how to handle them. Our approach to handling recursive modules follows that of MixML.

The example below defines two modules, A and B, which import each other. By forward-declaring the parts of B that A depends on, the first implementation makes sense—*i.e.*, it knows the names and types of entities it imports from B—and, naturally, the second implementation makes sense after that. This definition is analogous to how these modules would be defined in GHC today.[4]

```
package ab-rec where
  B :: [S_B]
  A = [import B; ...]
  B = [import A; ...]
```

Normal mixin linking ties the recursive knot, ensuring that the import B actually resolves to the B implementation in the end.

GHC allows recursive modules only within a single (Cabal) package. Backpack, on the other hand, allows more flexible recursion. Although packages themselves are not defined recursively, they may be recursively *linked*. Consider the following:

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

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

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

package ab-rec-sep where
  include a-from-b
  include b-from-a
```

[4] In GHC, instead of explicit bindings to a signature and two modules, there would be the two module source files and an additional "boot file" for B

that looks exactly like S_B. Moreover, the import B within the A module would include a "source pragma" that tells the compiler to import the boot file instead of the full module.

At the level of packages, these definitions do not involve any recursive inclusion, which is good, because that would be illegal! Rather, they form a diamond dependency, like the earlier packages top, left, right, and bottom. There is no recursion *within* the definitions of ab-sigs, a-from-b, and b-from-a either. The recursion instead occurs implicitly, as a result of the mixin linking of modules A and B in the package ab-rec-sep. (Separately typechecked, recursive units may be defined in MixML in roughly the same way.)

Finally, we note that Backpack's semantics (presented in the next section) explicitly addresses one of the key stumbling blocks in supporting recursive linking in the presence of abstract data types, namely the so-called *double vision problem* [6, 8]. In the context of the above example, the problem is that, in ab-sigs, the specification S_A of the hole A may specify an abstract type T, which S_B then depends on in the types of its core-level entities. Subsequently, in a-from-b, when the implementation of A imports B, it will want to know that the type T that it defines is the same as the one mentioned in S_B, or else it will suffer from "double vision", seeing two distinct names for the same underlying type. Avoiding double vision is known to be challenging [8, 9, 27], but crucial for enabling common patterns of recursive module programming. Backpack's semantics avoids double vision completely.

3. The Semantics of Backpack

The main top-level judgment defining the semantics of Backpack is

$$\Delta \vdash D : \forall \overline{\alpha}.\Xi \rightsquigarrow \lambda \overline{\alpha}.dexp$$

Given a *package definition* D, along with a *package environment* Δ describing the types and elaborations of other packages on which D depends, this judgment ascribes D a *package type* $\forall \overline{\alpha}.\Xi$, and also elaborates D into a *parameterized directory expression* $\lambda \overline{\alpha}.dexp$, which is essentially a set of well-typed Haskell module files.

The above judgment is implemented by a two-pass algorithm.[5] The first pass, called *shaping*, synthesizes a *package shape* $\tilde{\Xi}$ for D, which effectively explains the *macro-level* structure of the package, *i.e.*, the modules contained in D, the names of all the entities defined in those modules, and how they all depend on one another. The second pass, called *typing*, augments the structural information in $\tilde{\Xi}$ with additional information about the *micro-level* structure of D. In particular, it fills in the types of core-language entities, forming a *package type* Ξ and checking that D is well-formed at Ξ.

Our goal in the present section is to explain in detail this two-pass typechecking algorithm, as well as the elaboration of Backpack into Haskell. Central to both passes of Backpack typechecking is a notion of *module identity*. Using the multinst package (and its dependencies) from Figure 2 as a running example, we will motivate the role and structure of module identities, and then in subsequent subsections explain the implementation of shaping, typing, and elaboration. Full details are given in Appendix §6.

3.1 Module Identities

Figure 5 shows the *shapes* and *types* of multinst and its dependen-

cies. We proceed by explaining Figure 5 in a left-to-right fashion.

The first column of Figure 5 contains the first key component of package types: a mapping from modules' *logical* names p (*i.e.*, their names at the level of Backpack) to their *physical* identities ν (*i.e.*, the names of the Haskell modules to which they elaborate). The reason for distinguishing between logical names and physical identities is simple: due to aliasing (Section 2.5), there may be

[5] The reason for splitting typechecking into two passes has to do with the *double vision problem* [8], as discussed in Section 2.6. See Section 4 for further discussion, as well as a detailed explanation of how our solution to double vision compares with MixML's.

Identity Variables	$\alpha, \beta \in IdentVars$
Identity Constructors	$\mathcal{K} \in IdentCtors$
Identities	$\nu ::= \alpha \mid \mu\alpha.\mathcal{K}\,\overline{\nu}$
Identity Substitutions	$\phi, \theta ::= \{\overline{\alpha \coloneqq \nu}\}$

Figure 4. Module identities.

multiple logical names for the same physical module. (For further technical justifications for the logical/physical distinction, see the discussion in Section 4.)

In order to motivate the particular logical mappings in Figure 5, let us first explore what physical identities are, which means reviewing how module names work in Haskell.

Module Names in Haskell Modules in Haskell have fixed names, which we call "physical" because they are globally unique in a program, and module definitions may then depend on one another by importing these physical names. Modules serve two related roles: (1) as points of origin for core-level entities, and (2) as syntactic namespaces. Concerning (1), a module may *define* new

entities, such as values or abstract data types. Concerning (2), a module may *export* a set of entities, some of which it has defined itself and others of which it has imported from other modules. For example, a module Foo may define a data type named T. A subsequent module Bar may then import Foo.T and re-export it as Bar.T. To ensure that type identity is tracked properly, the Haskell type system models each core-level entity semantically as a pair $[\nu]\mathtt{T}$ of its core-level name T and its *provenance* ν, *i.e.*, the module that originally defined it (in this example, Foo). Thus, Foo.T and Bar.T will be viewed as equal by Haskell since they are both just different names for the same semantic entity [Foo]T.

To ensure compatibility with Haskell, our semantics for Backpack inherits Haskell's use of physical names to identify abstract types. However, Haskell's flat physical module namespace is not expressive enough to support Backpack's holes, applicative module instantiation, and recursive linking. To account for these features, we enrich the language of physical names with a bit more interesting structure. Figure 4 displays this enriched language of—as we call them—*physical module identities*.[6]

Variable and Applicative Identities Physical module identities ν are either (1) *variables* α, which are used to represent holes; (2) *applications* of identity *constructors* $\mathcal{K}$, which are used to model dependency of modules on one another, as needed to implement applicative instantiation; or (3) *recursive* module identities, defined via μ-constructors. We start by explaining the first two.

Each explicit module expression $[M]$ that occurs in a package definition corresponds (statically) to a globally unique identity constructor $\mathcal{K}$ that encodes it. For example, if a single module source M appears on the right-hand side of three distinct module bindings in a package P, then the three distinct identity constructors of those

modules are, roughly, $\langle P.M.1 \rangle$, $\langle P.M.2 \rangle$, and $\langle P.M.3 \rangle$.[7]

In the absence of recursive modules, each module identity ν is then a finite tree term—either a variable α, or a constructor $\mathcal{K}$ applied to zero or more subterms, $\overline{\nu}$. The identity of a module is the constructor $\mathcal{K}$ that encodes its source M, applied to the n identities to which M's n import statements resolved (in order). For instance, in the very first example from Section 2, ab-1, the identities of A and B are $\mathcal{K}_a$ and $\mathcal{K}_b\ \mathcal{K}_a$, respectively, where $\mathcal{K}_a$ encodes the first module expression and $\mathcal{K}_b$ the second. In a package with holes, each hole gets a fresh variable (within the package definition) as its

[6] To make use of these enriched physical names in our elaboration, we embed them into the space of Haskell's physical names; see Section 3.4.

[7] We write simply $\langle P.M \rangle$, eliding the integer part of the identity constructor, when only one instance of $[M]$ exists in the definition of package P.

identity; in abcd-holes-1 the identities of the four modules are, in order, α_a, α_b, $\mathcal{K}_c$, and $\mathcal{K}_d\ \alpha_a\ \mathcal{K}_c$.

Consider now the module identities in the Graph instantiations in multinst, as shown in Figure 5. In the definition of structures, assume that the variables for Prelude and Array are α_P and α_A respectively, and that M_G is the module source that Graph is bound to. Then the identity of Graph is $\nu_G = \langle \mathsf{structures}.M_G \rangle\ \alpha_P\ \alpha_A$. Similarly, the identities of the two array implementations in Figure 2 are $\nu_{AA} = \langle \mathsf{arrays\text{-}a}.M_A \rangle\ \alpha_P$ and $\nu_{AB} = \langle \mathsf{arrays\text{-}b}.M_B \rangle\ \alpha_P$.

The package graph-a is more interesting because it *links*

the packages arrays-a and structures together, with the implementation of Array from arrays-a *instantiating* the hole Array from structures. This linking is reflected in the identity of the Graph module in graph-a: whereas in structures it was $\nu_G = \langle \mathsf{structures}.M_G \rangle\ \alpha_P\ \alpha_A$, in graph-a it is $\nu_{GA} = \nu_G[\nu_{AA}/\alpha_A] = \langle \mathsf{structures}.M_G \rangle\ \alpha_P\ \nu_{AA}$. Similarly, the identity of Graph in graph-b is $\nu_{GB} = \nu_G[\nu_{AB}/\alpha_A] = \langle \mathsf{structures}.M_G \rangle\ \alpha_P\ \nu_{AB}$. Thus, linking consists of substituting the variable identity of a hole by the concrete identity of the module filling that hole.

Lastly, multinst makes use of both of these Graph modules, under the aliases GA and GB, respectively. Consequently, in the Client module, GA.G and GB.G will be correctly viewed as distinct types since they originate in modules with distinct identities.

As multinst illustrates, module identities effectively encode dependency graphs. The primary motivation for encoding this information in identities is our desire for an *applicative* semantics of instantiation, as needed for instance in the example of Figure 3. In that example, both the packages applic-left and applic-right individually link arrays-a with structures. The client package applic-bot subsequently wishes to use both the Left module from applic-left and the Right module from applic-right, and depends on the fact that both modules operate over the same Graph.G type. This fact will be checked when the packages applic-left and applic-right are both **include**d in the same namespace of applic-bot, and the semantics of mixin linking will insist that their Graph modules have the same identity. Thanks to the dependency tracking in our module identities, we know that the Graph module has identity ν_{GA} in both packages.

Recursive Module Identities In the presence of recursive mod-

ules, module identities are no longer simple finite trees.

Consider again the ab-rec-sep example from the end of Section 2. (Although this example does not concern our current focus, multinst, the careful treatment of recursive module identities deserves explanation.) Suppose that ν_A and ν_B are the identities of A and B, and that M_A and M_B are those modules' defining module expressions, respectively. Because M_A imports B and M_B imports A, the two identities should satisfy the recursive equations

$$\nu_A = \langle \text{a-from-b}.M_A \rangle\ \nu_B$$
$$\nu_B = \langle \text{b-from-a}.M_B \rangle\ \nu_A$$

These identity equations have no solution in the domain of finite trees, but they do in the domain of regular, *infinite* trees, which we denote (finitely) as

$$\nu_A = \mu\alpha_A.\langle \text{a-from-b}.M_A \rangle\ (\langle \text{b-from-a}.M_B \rangle\ \alpha_A)$$
$$\nu_B = \mu\alpha_B.\langle \text{b-from-a}.M_B \rangle\ (\langle \text{a-from-b}.M_A \rangle\ \alpha_B)$$

The semantics of Backpack relies on the ability to perform both *unification* and *equivalence* testing on identities. In the presence of recursive identities, however, simple unification and syntactic equivalence of identities no longer suffices since, *e.g.*, the identity $\langle \text{a-from-b}.M_A \rangle\ \nu_B$ represents the exact same module as ν_A, albeit in a syntactically distinct way. Fortunately, we can use Huet's well-known unification algorithm for regular trees instead [18, 14].

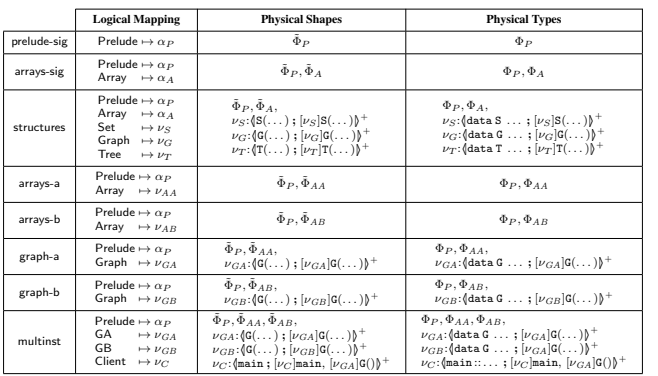

Figure 5. Example package types and shapes for the multinst package and its dependencies.

3.2 Shaping

Constructing the mapping from logical names to physical identities is but one part of a larger task we call *shaping*, which constitutes the most unusual and interesting part of Backpack's type system.

The goal of shaping is to compute the shape (*i.e.*, the macro-

level structure) of the package. Formally, a package shape $\tilde{\Xi} = (\tilde{\Phi}; \tilde{\mathcal{L}})$ has two parts.[8] The first is a *physical shape context* $\tilde{\Phi} = \overline{\nu{:}\tilde{\tau}^m}$, which, for each module in the package, maps its physical identity ν to a *polarity* m and a *module shape* $\tilde{\tau}$. The polarity m specifies whether the module ν is implemented $(+)$ or a hole $(-)$. The module shape $\tilde{\tau} = \langle\!|\, \overline{dent} \;;\; \overline{espc} \,|\!\rangle$ enumerates ν's *defined entities* $\overline{dent}$—*i.e.*, the entities that the module ν itself defines—as

[8] We write a tilde $(\tilde{\cdot})$ on the metavariables of certain shape objects (*e.g.*, $\tilde{\tau}$) not to denote a meta-level operation, but to highlight these objects' similarity to their corresponding type objects (*e.g.*, τ).

well as *export specs* $\overline{espc}$, which list the names and provenances of the entities that ν exports. Note that these are not the same thing: a module ν may import and re-export entities that originated in (*i.e.*, whose provenances are) some other modules $\overline{\nu'}$, and it may also choose *not* to export all of the entities that it defines internally. In our running example in Figure 5, the physical shape contexts $\tilde{\Phi}$ computed for each package are shown in the second column.

The second part of the package shape is a *logical shape context* $\tilde{\mathcal{L}} = \overline{p \mapsto \nu @ \tilde{\tau}}$, which, for each module in the package, maps its logical name p to its physical identity ν. (This is the mapping shown in the first column of Figure 5, which we have already discussed in detail in Section 3.1). In addition, $\tilde{\mathcal{L}}$ also maps p to a shape $\tilde{\tau}$, which is required to be a subset of the "principal shape" of ν (as recorded in the physical shape context $\tilde{\Phi}$). This may seem

mysterious: if $\mathcal{L}$ maps p to ν, and Φ maps ν to $\tilde{\tau}_0$, say, then why does $\tilde{\mathcal{L}}$ record another $\tilde{\tau}$ as well? The reason is twofold: (1) it is

Core Level:			
Value Names	x	$\in$	$ValNames$
Type Names	T	$\in$	$TypeNames$
Constructor Names	K	$\in$	$CtorNames$
Kind Environments	$kenv$	$::=$	$\ldots$
Semantic Types	typ	$::=$	$[\nu]T\ \overline{typ} \mid \ldots$
Defined Entity Specs	$dspc$	$::=$	$\texttt{data}\ T\ kenv = \overline{K\ \overline{typ}}$
		$\mid$	$\texttt{data}\ T\ kenv \quad \mid \quad x :: typ$
Export Specs	$espc$	$::=$	$[\nu]dent$
Module Level:			
Polarities	m	$::=$	$+ \mid -$
Types	τ, σ	$::=$	$\langle\!\langle \overline{dspc}\ ;\ \overline{espc} \rangle\!\rangle$
Physical Type Ctxts	Φ	$::=$	$\overline{\nu{:}\tau^m}$
Logical Type Ctxts	$\mathcal{L}$	$::=$	$\overline{p \mapsto \nu@\tau}$
Package Level:			
Package Types	Ξ, Γ	$::=$	$(\Phi; \mathcal{L})$
Package Environments	Δ	$::=$	$\cdot \mid \Delta,\ P = \lambda\overline{\alpha}.dexp : \forall\overline{\alpha}.\Xi$
Shaping Objects:			
Defined Entities	$dent$	$::=$	$x \mid T \mid T(\overline{K})$
Module Shapes	$\tilde{\tau}$	$::=$	$\langle\!\langle \overline{dent}\ ;\ \overline{espc} \rangle\!\rangle$
Physical Shape Ctxts	$\tilde{\Phi}$	$::=$	$\overline{\nu{:}\tilde{\tau}^m}$
Logical Shape Ctxts	$\tilde{\mathcal{L}}$	$::=$	$\overline{p \mapsto \nu@\tilde{\tau}}$
Package Shapes	$\tilde{\Xi}, \tilde{\Gamma}$	$::=$	$(\tilde{\Phi}; \tilde{\mathcal{L}})$

Figure 6. Semantic objects for shaping and typing.

convenient in some of the inference rules to be able to look up the shape of a module by merely inspecting the logical shape context $\tilde{\mathcal{L}}$, and (2) it is possible for different logical module names to reflect different restricted subnamespaces of the same underlying module

(see the technical appendix for an example of this; Appendix §3). In our running example, however, the reader may ignore this subtlety and assume that all the $\tilde{\tau}$'s associated with ν in $\tilde{\mathcal{L}}$ and $\tilde{\Phi}$ are equal (which is why we omit them from the first column of Figure 5).

Figure 6 defines the semantic objects for shaping and typing, and Figure 7 gives some of the key rules implementing shaping.

Shaping Rules The main shaping judgment, $\Delta \Vdash \overline{B} \Rightarrow \tilde{\Xi}$, takes as input the body of a package definition, which is just a sequence of bindings $\overline{B}$. Rule SHSEQ synthesizes the shape of $\overline{B}$ by proceeding, in left-to-right order, to synthesize the shape of each individual binding B (via the judgment $\Delta; \tilde{\Gamma} \Vdash B \Rightarrow \tilde{\Xi}$) and then *merge* it with the shapes of the previous bindings (via the judgment $\Vdash \tilde{\Xi}_1 + \tilde{\Xi}_2 \Rightarrow \tilde{\Xi}$).

Let us begin with the judgment that shapes an individual binding. The rule SHALIAS should be self-explanatory.

The rule SHINC is simple as well, choosing fresh identity variables $\overline{\alpha}$ to represent the holes in package P and applying the renaming r to P's shape. (Note that it uses some simple auxiliary definitions: rename, for applying a renaming to the $\mathcal{L}$ part of a shape, and shape, for erasing a package type Ξ to a shape by removing typing information. Moreover, by alpha-varying the type of P we rename its variables to match the freshly chosen $\overline{\alpha}$.)

The rule SHMOD generates the appropriate globally unique identity ν_0 to represent $[M]$, and then calls out to a shaping judgment for Haskell modules, $\tilde{\Gamma}; \nu_0 \Vdash_{\mathsf{c}} M : \tilde{\tau}$ (Appendix §5.1.1), which generates the shape $\tilde{\tau}$ of M assuming that ν_0 is the module's identity. As an example of this, observe the shape generated for the Client module ν_C in multinst in Figure 5. The shape ascribes provenance ν_C to the `main` entity, since it is freshly defined in Client, while ascribing provenance ν_{GA} to the `G` type, since it

was imported from GA and is only being re-exported by Client.

The rule SHSIG, for shaping hole declarations, is a bit subtler than the other rules. Perhaps surprisingly, the generated shape declares not only a fresh identity variable α for the hole itself, but also a set of fresh identity variables $\overline{\beta}$, one for each entity speci-

$$\boxed{\Delta; \tilde{\Gamma} \Vdash B \Rightarrow \tilde{\Xi}} \qquad \frac{p' \mapsto \nu @ \tilde{\tau} \in \tilde{\Gamma}}{\Delta; \tilde{\Gamma} \Vdash p = p' \Rightarrow (\emptyset; p \mapsto \nu @ \tilde{\tau})} \ (\textsc{ShAlias})$$

$$\frac{\nu_0 = \mathsf{mkident}(M; \tilde{\Gamma}) \qquad \tilde{\Gamma}; \nu_0 \Vdash_{\mathrm{c}} M : \tilde{\tau}}{\Delta; \tilde{\Gamma} \Vdash p = [M] \Rightarrow (\nu_0 : \tilde{\tau}^+; p \mapsto \nu_0 @ \tilde{\tau})} \ (\textsc{ShMod})$$

$$\frac{\alpha, \overline{\beta}\ \mathsf{fresh} \qquad \tilde{\tau}_0 = \langle\!\langle \emptyset \ ; \ \overline{[\beta] dent} \rangle\!\rangle \qquad \tilde{\Gamma}; \tilde{\tau}_0 \Vdash_{\mathrm{c}} S \rightsquigarrow \tilde{\tau}; \tilde{\Phi}'}{\Delta; \tilde{\Gamma} \Vdash p :: [S] \Rightarrow ((\alpha : \tilde{\tau}^-, \tilde{\Phi}'); p \mapsto \alpha @ \tilde{\tau})} \ (\textsc{ShSig})$$

$$\frac{\overline{\alpha}\ \mathsf{fresh} \qquad (P : \forall \overline{\alpha}. \Xi) \in \Delta \qquad \Xi' = \mathsf{rename}(r; \Xi)}{\Delta; \tilde{\Gamma} \Vdash \textbf{include}\ P\ r \Rightarrow \mathsf{shape}(\Xi')} \ (\textsc{ShInc})$$

$$\boxed{\Delta \Vdash \overline{B} \Rightarrow \tilde{\Xi}} \qquad \frac{}{\Delta \Vdash \emptyset \Rightarrow (\emptyset; \emptyset)} \ (\textsc{ShNil})$$

$$\frac{\Delta \Vdash \overline{B_1} \Rightarrow \tilde{\Xi}_1 \qquad \Delta; \tilde{\Xi}_1 \Vdash B_2 \Rightarrow \tilde{\Xi}_2 \qquad \Vdash \tilde{\Xi}_1 + \tilde{\Xi}_2 \Rightarrow \tilde{\Xi}}{\Delta \Vdash \overline{B_1}, B_2 \Rightarrow \tilde{\Xi}} \ (\textsc{ShSeq})$$

Figure 7. Shaping rules (ignoring thinning).

fied in the hole signature S. (The intermediate $\tilde{\tau}_0$ merely encodes

these fresh identities as input to the signature shaping judgment.) The reason for this is simply to maximize flexibility: there is no reason to demand *a priori* that the module that fills in the hole (*i.e.*, the module whose identity ν will end up getting substituted for α) must itself be responsible for *defining* all the entities specified in the hole signature—it need only be responsible for *exporting* those entities, which may very well have been defined in other modules.

The shape $\tilde{\Phi}_A$ in Figure 5 illustrates the output of SHSIG on the Array hole in package arrays-sig. This shape specifies that β_{AA} is a module defining an entity called `Arr`, that β_{AS} is a module defining an entity called `something`, and that α_A is a module bringing $[\beta_{AA}]$`Arr` and $[\beta_{AS}]$`something` together in its export spec. Of course, when the hole is eventually filled (*e.g.*, in the graph-a package, whose shaping is discussed below), it may indeed be the case that the same module identity ν is substituted for α_A, β_{AA}, and β_{AS}—*i.e.*, that ν both defines *and* exports `Arr` and `something`—but SHSIG does not require this.

Returning now to the merging judgment $\Vdash \tilde{\Xi}_1 + \tilde{\Xi}_2 \Rightarrow \tilde{\Xi}$ that is invoked in the last premise of (SHSEQ): This merging judgment (Appendix §6.6) is where the real "meat" of shaping occurs—in particular, this is where mixin *linking* is performed by *unification* of module identities. If a module with logical name p is mapped by $\tilde{\Xi}_1$ and $\tilde{\Xi}_2$ to physical identities ν_1 and ν_2, respectively, the merging judgment will unify ν_1 and ν_2 together. Moreover, if ν_1 and ν_2 are specified by $\tilde{\Xi}_1$ and $\tilde{\Xi}_2$ as having different module shapes $\tilde{\tau}_1$ and $\tilde{\tau}_2$, respectively, those shapes will be merged as well, with the resulting shape containing all of the components specified in either $\tilde{\tau}_1$ and $\tilde{\tau}_2$. For any entities appearing in both $\tilde{\tau}_1$ and $\tilde{\tau}_2$, their provenances will be unified.

To see a concrete instance of this, consider the merging that occurs during the shaping of the graph-a package in our running example in Figure 5. The graph-a package **include**s two packages defined earlier: arrays-a and structures. As per rule (SHINC), each inclusion will generate fresh identity variables for the packages' holes (say, α_P, β_{PL}, α_A, β_{AA}, β_{AS} for structures, and α'_P, β'_{PL} for arrays-a). Since both packages export Prelude, the merging judgment will unify α_P and α'_P, the physical identities associated with Prelude in the shapes of the two packages; consequently, the shape of α_P, namely $\langle\!|\emptyset \ ; [\beta_{PL}]\texttt{List}(\texttt{Nil}, \texttt{Cons})|\!\rangle$, will be unified with the shape of α'_P, namely $\langle\!|\emptyset \ ; [\beta'_{PL}]\texttt{List}(\texttt{Nil}, \texttt{Cons})|\!\rangle$, resulting in the unification of β_{PL} and β'_{PL} as well.

Similarly, since both packages export Array, the merging judgment will link the implementation of Array in arrays-a with the hole for Array in structures by unifying α_A, β_{AA}, and β_{AS} with

ν_{AA}. As a result, the occurrences of α_A, β_{AA}, and β_{AS} in ν_G (and its shape) get substituted with ν_{AA}, which explains why the shape of graph-a maps Graph to $\nu_{GA} = \nu_G[\nu_{AA}/\alpha_A]$. Lastly, merging will check that the implementation of Array in arrays-a actually provides all the entities required by the hole specification in structures, *i.e.*, that $\tilde{\Phi}_{AA}$ subsumes $\tilde{\Phi}_A$, which indeed it does.

3.3 Typing

In our running example thus far, we have not yet performed any *typechecking* of core-level code, such as the code inside multinst's Client module. There is a good reason for this: before shaping, we don't know whether core-level types such as GA.G and GB.G (imported by Client) are equal, because we don't know what the identities of GA and GB are. But after shaping, we have all the identity information we need to perform typechecking proper.

Thus, as seen in the top-level package rule TyPkg in Figure 8, the output of the shaping judgment—namely, $\tilde{\Xi}_{\mathsf{pkg}}$—is passed as input to the *typing* judgment, $\Delta; \tilde{\Xi}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \rightsquigarrow \mathit{dexp}$. Typing, in turn, produces a *package type* Ξ, which enriches the package shape $\tilde{\Xi}_{\mathsf{pkg}}$ with core-level (*i.e.*, Haskell-level) typing information. The final type returned for the package, $\forall\overline{\alpha}.\Xi$, then just quantifies over the variable identities $\overline{\alpha}$ of the holes in Ξ, so that they may be instantiated in different ways by subsequent package definitions.

The package types Ξ generated for the packages in our running example appear in the third column of Figure 5. Formally, the only difference between these package types and the package shapes in the second column of Figure 5 lies in the difference between their constituent *module* types $\tau = \langle\!|\, \overline{dspc} \,;\, \overline{espc} \,|\!\rangle$ and *module* shapes $\tilde{\tau} = \langle\!|\, \overline{dent} \,;\, \overline{espc} \,|\!\rangle$. Whereas the "defined entities" component ($\overline{dent}$) of $\tilde{\tau}$ only *names* the entities defined by a module, the

"defined entity specs" component ($\overline{dspc}$) of τ additionally specifies their core-level kinds/types. For example, observe the module type ascribed to arrays-a's module ν_{AA} in Φ_{AA}. This type enriches the pre-computed shape (in $\tilde{\Phi}_{AA}$) with additional information about the kind of `Arr` and the type of `something`.

Let us now explain the typing rules in Figure 8. For the moment, we will ignore the shaded parts of the rules concerning elaboration into Haskell; we will return to them in Section 3.4.

The rules TyNil and TySeq implement typing of a sequence of bindings $\overline{B}$. The procedure is structurally very similar to the one used in the shaping of $\overline{B}$: we process (in left-to-right order) each constituent binding B, producing a *type* that we *merge* into the types of the previous bindings. The key difference is that the partial merge operator $\oplus$ does not perform any unification on module identities—it merely performs a mixin merge, which checks that all specifications (kinds or types) assigned to any particular core-level entity are equal. For instance, when typing graph-a, the mixin merge will check that the type of `something` in the Array implementation from arrays-a is equal to the type of `something` in the Array hole from structures, and thus that the implementation satisfies the requirements of the hole.

The remaining rules concern the typing of individual bindings, $\Delta; \Gamma; \tilde{\Xi}_{\mathsf{pkg}} \vdash B : \Xi \rightsquigarrow dexp$. The typing rules TyMod and TySig are structurally very similar to the corresponding shaping rules given in Figure 7. The key difference is that, whereas ShMod and ShSig *generate* appropriate identities for their module/hole, TyMod and TySig instead *look up* the pre-computed identities in the package shape $\tilde{\Xi}_{\mathsf{pkg}}$. As an example of this, observe what happens when we type the Array module in arrays-a using rule TyMod. The package shape $\tilde{\Xi}_{\mathsf{pkg}}$ we pre-computed in the shap-

ing pass tells us that the physical module identity associated with the logical module name Array is ν_{AA}, so we can go ahead and assume ν_{AA} is the identity of Array when typing its implementa-

$\boxed{\Delta; \Gamma; \tilde{\Xi}_{\mathsf{pkg}} \vdash B : \Xi \rightsquigarrow \mathit{dexp}}$

$$\frac{p' \mapsto \nu @ \tau \in \Gamma}{\Delta; \Gamma; \tilde{\Xi}_{\mathsf{pkg}} \vdash p = p' : (\emptyset; p \mapsto \nu @ \tau) \rightsquigarrow \{\}} \ (\textsc{TyAlias})$$

$$\frac{p \mapsto \nu_0 @ \tilde{\tau}_0 \in \tilde{\Xi}_{\mathsf{pkg}} \qquad \Gamma; \nu_0 \vdash_{\mathrm{c}} M : \tau \rightsquigarrow \mathit{hsmod}}{\begin{array}{c}\Delta; \Gamma; \tilde{\Xi}_{\mathsf{pkg}} \vdash p = [M] : (\nu_0{:}\tau^+; p \mapsto \nu_0 @ \tau) \\ \rightsquigarrow \ \{\nu_0{}^\star \mapsto \mathit{hsmod} : \tau^\star\}\end{array}} \ (\textsc{TyMod})$$

$$\frac{\begin{array}{c}p \mapsto \nu_0 @ \tilde{\tau}_0 \in \tilde{\Xi}_{\mathsf{pkg}} \qquad \Gamma; \tilde{\tau}_0 \vdash_{\mathrm{c}} S \rightsquigarrow \tau; \Phi' \\ \Phi'' = \nu_0{:}\tau^- \oplus \Phi' \text{ defined}\end{array}}{\begin{array}{c}\Delta; \Gamma; \tilde{\Xi}_{\mathsf{pkg}} \vdash p :: [S] : (\Phi''; p \mapsto \nu_0 @ \tau) \\ \rightsquigarrow \ \{\nu^\star \mapsto - : \sigma^\star \mid \nu{:}\sigma^- \in \Phi''\}\end{array}} \ (\textsc{TySig})$$

$$\frac{\begin{array}{c}\overline{\alpha} \text{ fresh} \quad (P = \lambda \overline{\alpha}.\mathit{dexp} : \forall \overline{\alpha}.\Xi) \in \Delta \quad \Xi' = \mathsf{rename}(r; \Xi) \\ \vdash \tilde{\Xi}_{\mathsf{pkg}} \leq_{\overline{\alpha}} \Xi' \rightsquigarrow \phi \qquad \Xi'' = \mathsf{apply}(\phi; \Xi') \text{ defined}\end{array}}{\Delta; \Gamma; \tilde{\Xi}_{\mathsf{pkg}} \vdash \textbf{include}\ P\ r : \Xi'' \rightsquigarrow \mathsf{apply}(\phi^\star; \mathit{dexp})} \ (\textsc{TyInc})$$

$\boxed{\Delta; \tilde{\Xi}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \rightsquigarrow \mathit{dexp}}$

$$\frac{}{\Delta; \tilde{\Xi}_{\mathsf{pkg}} \vdash \emptyset : (\emptyset; \emptyset) \rightsquigarrow \{\}} \ (\textsc{TyNil})$$

$$\frac{\Delta; \tilde{\Xi}_{\mathsf{pkg}} \vdash \overline{B_1} : \Xi_1 \rightsquigarrow dexp_1 \qquad \Xi = \Xi_1 \oplus \Xi_2 \text{ defined} \qquad \Delta; \Xi_1; \tilde{\Xi}_{\mathsf{pkg}} \vdash B_2 : \Xi_2 \rightsquigarrow dexp_2}{\Delta; \tilde{\Xi}_{\mathsf{pkg}} \vdash \overline{B_1}, B_2 : \Xi \rightsquigarrow dexp_1 \oplus dexp_2} \text{ (TySeq)}$$

$$\boxed{\Delta \vdash D : \forall \overline{\alpha}.\Xi \rightsquigarrow \lambda \overline{\alpha}.dexp}$$

$$\frac{\Delta \Vdash \overline{B} \Rightarrow \tilde{\Xi}_{\mathsf{pkg}} \qquad \Delta; \tilde{\Xi}_{\mathsf{pkg}} \vdash \overline{B} : \Xi \rightsquigarrow dexp \qquad \overline{\alpha} = \mathsf{fv}(\Xi)}{\Delta \vdash \textbf{package } P \textbf{ where } \overline{B} : \forall \overline{\alpha}.\Xi \rightsquigarrow \lambda \overline{\alpha}.dexp} \text{ (TyPkg)}$$

Figure 8. Typing and elaboration rules (ignoring thinning).

tion. Note that TyMod and TySig call out to typing judgments for Haskell modules and signatures ($\vdash_{\mathrm{c}}$). Like the analogous shaping judgments, these are defined formally in the appendix (Appendix §5.1.1).

Like TyMod and TySig, the rule TyInc also inspects $\tilde{\Xi}_{\mathsf{pkg}}$ to determine the pre-computed identities of the modules/holes in the package P being **include**d. The only difference is that an **include**d package contains a whole bunch of subcomponents (rather than only one), so looking up their identities is a bit more involved. It is performed by appealing to a "matching" judgment $\vdash \tilde{\Xi}_{\mathsf{pkg}} \leq_{\overline{\alpha}} \Xi' \rightsquigarrow \phi$, similar to the one needed for signature matching in ML module systems [28]. This judgment looks up the instantiations of all the **include**d holes by matching Ξ' (the type of the **include**d package P *after* applying the renaming r) against $\tilde{\Xi}_{\mathsf{pkg}}$.

This produces a substitution ϕ with domain $\overline{\alpha}$, which then gets applied to Ξ' to produce the type of the **include** binding. For example, when typing the package graph-a, we know after shaping that the identity of the Array module is ν_{AA}. When we **include** structures, the matching judgment will glean this information from $\widetilde{\Xi}_{\mathsf{pkg}}$, and produce a substitution ϕ mapping structures' α_A parameter to the actual Array implementation ν_{AA}.

3.4 Elaborating Backpack to Haskell

We substantiate our claim to retrofit Haskell with SMD through an elaboration of Backpack, our external language (EL), into a model of GHC Haskell, our internal language (IL). The EL, as we have

$$
\begin{array}{lrl}
\text{(Module Names)} & f & \in \mathit{IlModNames} \\
\text{(Module Sources)} & \mathit{hsmod} & ::= \ldots \\
\text{(File Expressions)} & \mathit{fexp} & ::= \mathit{hsmod} \mid - \\
\text{(File Types)} & \mathit{ftyp} & ::= \langle\!| \overline{\mathit{dspc}^\star} \mathbin{;} \overline{\mathit{espc}^\star} |\!\rangle \\
\text{(Typed File Expressions)} & \mathit{tfexp} & ::= \mathit{fexp} : \mathit{ftyp} \\
\text{(Directory Expressions)} & \mathit{dexp} & ::= \{\overline{f \mapsto \mathit{tfexp}}\} \\
\text{(Identity Translation)} & (-)^\star & \in \mathit{Identities}/{\equiv_\mu} \rightarrowtail \mathit{IlModNames}
\end{array}
$$

Figure 9. IL syntax. ($dspc^\star$ and $espc^\star$ mention f instead of ν.)

demonstrated so far, extends across the package, module, and core levels, while the IL defines only module and core levels; effectively the outer, package level gets "compiled away" into mere modules in the IL. Figure 9 gives the syntax of the IL; for its semantics, including the typing judgment, see the technical appendix (Appendix §7).

Elaboration translates a Backpack package into a *parameterized directory expression* $\lambda\overline{\alpha}.\mathit{dexp}$, which is a mapping from a set of module names f to typed file expressions tfexp, parameterized over the identities $\overline{\alpha}$ of the package's holes. We assume an embedding $(-)^\star$ from module identities into IL module names, which respects the equi-recursive equivalence on module identities that the Backpack type system relies on. However, for readability, we will leave the embedding implicit in the remainder of this section. As for the typed file expressions tfexp, they can either be defined file expressions ($\mathit{hsmod} : \mathit{ftyp}$), which provide both an implementation of a module along with its type, or undefined file expressions ($- : \mathit{ftyp}$), which describe a hole with type ftyp. Thus, all components of a dexp are explicitly-typed. This has the benefit that the modules in a dexp can be typechecked in any order, since all static information about them is specified in their explicit file types.

As a continuation of our running example, Figure 10 displays

the elaboration of the multinst package, except with the file types stripped off for brevity. First, note that each module identity ν in the physical type Φ_M of multinst (lower-right hand corner of the table in Figure 5) corresponds to one of the Haskell modules in the elaboration of the package, and for each ν, its type in Φ_M is (modulo the embedding $(-)^\star$) precisely the file type of ν that we have omitted from Figure 10. The concrete module identities in Φ_M map to defined file expressions, while the identity variables α_P and β_{PL} (representing holes) map to undefined file expressions.

The elaboration of packages (marked with shaded text) is almost entirely straightforward, following the typing rules. More interesting is the elaboration of Haskell *modules*, which is appealed to in the second premise of rule TyMod (and formalized in the appendix; Appendix §6). Offhand, one might expect module elaboration to be the identity translation, but in fact it is a bit more subtle than that.

Consider the ν_C entry in the directory, corresponding to the Client module, as a concrete example.

The module header: Unlike the original EL implementation of Client, which was anonymous, its elaborated IL version has a module header specifying ν_C as its fixed physical name, and `main` and `GA.G()` as its exported entities. More generally, the exported entities should reflect those listed in the module's type τ.

The import statements: Our elaboration rewrites imports of logical names like GA into imports of physical module identities like ν_{GA}, since the physical identities are the actual names of Haskell modules in the elaborated directory expression. We must therefore take care to preserve the logical module names that the definitions in the module's body expect to be able to refer to. For example, the reference GA.G is seen to have provenance $[\nu_{GA}]$G during Backpack typechecking of Client, so in the elaborated IL version of Client we

want GA.G to mean the same thing. We achieve this by means of Haskell's "import aliases", which support renaming of imported module names; *e.g.*, the first import statement in ν_C imports the

$$
\lambda \alpha_P\, \beta_{PL}\,.\;
\left\{
\begin{array}{lcl}
\alpha_P & \mapsto & \text{———————————} \\
\beta_{PL} & \mapsto & \text{———————————} \\
\nu_{AA} & \mapsto & \left(\begin{array}{l}\texttt{module } \nu_{AA}\ \texttt{(Arr(MkArr))}\ \texttt{where} \\ \quad \texttt{import qualified } \alpha_P \texttt{ as P (List(Nil,Cons))} \\ \quad \texttt{data Arr i e = MkArr ...} \\ \quad \texttt{something = P.Nil :: P.List (Arr i e)}\end{array}\right) \\
\nu_{AB} & \mapsto & \left(\begin{array}{l}\texttt{module } \nu_{AB}\ \texttt{(Arr(ANil, ...))}\ \texttt{where} \\ \quad \texttt{import } \alpha_P \texttt{ as Prelude (List(Nil, Cons))} \\ \quad \texttt{data Arr i e = ANil | ...} \\ \quad \texttt{something = Cons ANil Nil}\end{array}\right) \\
\nu_{GA} & \mapsto & \left(\begin{array}{l}\texttt{module } \nu_{GA}\ \texttt{(G(...))}\ \texttt{where} \\ \quad \texttt{import } \alpha_P \texttt{ as Prelude (List(Nil, Cons))} \\ \quad \texttt{import } \nu_{AA} \texttt{ as Array (Arr(), something)} \\ \quad \texttt{data G ...}\end{array}\right) \\
\nu_{GB} & \mapsto & \left(\begin{array}{l}\texttt{module } \nu_{GB}\ \texttt{(G(...))}\ \texttt{where} \\ \quad \texttt{import } \alpha_P \texttt{ as Prelude (List(Nil, Cons))} \\ \quad \texttt{import } \nu_{AB} \texttt{ as Array (Arr(), something)} \\ \quad \texttt{data G ...}\end{array}\right) \\
\nu_C & \mapsto & \left(\begin{array}{l}\texttt{module } \nu_C\ \texttt{(main, GA.G())}\ \texttt{where} \\ \quad \texttt{import qualified } \nu_{GA} \texttt{ as GA (G())} \\ \quad \texttt{import qualified } \nu_{GB} \texttt{ as GB (G())} \\ \quad \texttt{main = ... GA.G ... GB.G ...}\end{array}\right)
\end{array}
\right\}
$$

Figure 10. Elaboration of multinst. (For readability, the translation from identities to module names, $(-)^\star$, and the file type annotation on each module file have been omitted. See Figure 5 for the latter.)

physical name ν_{GA} but gives it the logical name GA in the body, thus ensuring that the reference GA.G still has (the same) meaning as it did during Backpack typechecking.

The body: Thanks to the import aliasing we just described, the entity definitions in the body of ν_C can remain syntactically identical to those in the original Client module.

Explicitness of imports and exports: All imported and exported entities are given as explicitly as the Haskell module syntax allows, even when the original EL modules neglect to make them explicit; *e.g.*, the original code for Graph lists neither its imports nor its exports, but its elaboration (as ν_{GA} and ν_{GB}) does. The primary reason for this explicitness is that it enables us to prove a "weakening" property on IL modules, which is critical for the proof of soundness of elaboration. If modules are not explicit about which core-level entities they are importing and exporting, their module types will not be stable under weakening.

3.5 Formalization of Haskell Modules and Soundness

We have proven the following key soundness theorem about the elaboration: if a package definition D has type $\forall\overline{\alpha}.(\Phi;\mathcal{L})$ and elaborates into a parameterized directory expression $\lambda\overline{\alpha}.dexp$, then every module in $dexp$ is well-typed in the IL, and the identities and types in Φ directly match those of $dexp$. (For example, soundness

means that the identities and EL module types appearing in the type of multinst in Figure 5 correspond precisely to the names and types of the IL modules in Figure 10.)

As part of this effort, our IL constitutes a new formal model of the Haskell module system. This model follows the Haskell Language Report as closely as possible in its definition of well-formed modules—*i.e.*, the processing of module dependencies through import statements and export lists. Unlike previous work on formalizing the Haskell module system [7, 12], our model supports (separately typechecked) recursive modules; furthermore, we have developed some basic metatheory for the IL (*e.g.*, "substitution" and "weakening") in order to prove soundness of elaboration, a non-

trivial undertaking given that substitution can result in the merging of module/signature bindings. Full details of the IL are given in the accompanying technical appendix (Appendix §7).

We do not provide any kind of dynamic semantics in Backpack and thus we cannot prove (or even state) any conventional type safety theorem. Instead, the soundness of Backpack's elaboration simply reduces the question of whether Backpack is type-safe to the question of whether Haskell is type-safe.

4. Related Work and Technical Discussion

Detailed Comparison with MixML ML modules provide a very expressive and convenient language for programming with abstract data types. However, due to the double vision problem (Section 2.6), functors are fundamentally incompatible with recursive linking [27]. There have therefore been several attempts to synthesize aspects of ML modules and mixin modules in a single system, including Owens and Flatt's typed unit calculus [25] and Duggan's type system for recursive DLLs [10].

Arguably the most advanced system in this space is Rossberg and Dreyer's MixML [27], which aims to be a highly expressive foundational calculus of mixin modules for an ML-like core language. Backpack's design and semantics are inspired by those of MixML, but our design decisions, driven by our goal of retrofitting Haskell with SMD, have led to considerable simplifications.

MixML supports first-class and higher-order *units* (*i.e.*, instantiable mixins), whereas Backpack's units—packages—only exist at the top level. We believe Backpack's units to be sufficient for practical programming, and restricting them to top level streamlines the semantics of applicative identities. MixML also supports hierarchical mixin modules with "deep linking", but Backpack restricts packages to be flat namespaces of modules. Deep linking

lets MixML express many different ML constructs (*e.g.*, n-ary signatures) using just one form of linking. Since our focus is on practicality rather than expressiveness, we sacrifice features like first-class units and deep linking for simplicity of syntax and semantics, optimizing instead for common usage patterns. In particular, our **include** construct is more straightforward to program with than MixML's binary linking/binding construct, and fits better with the "feel" of the Haskell module language (*e.g.*, its **import** statements). Backpack also provides some features that MixML lacks, such as renaming, thinning, and an applicative semantics of instantiation.

Concerning the double vision problem, MixML solves it through the use of a two-pass algorithm for typechecking linked modules: the first pass computes all information about type components in the modules, and the second pass performs full typechecking. In MixML, these two passes are defined using a single set of inference rules, with the first pass defined by conveniently ignoring certain premises. Backpack adopts the same two-pass idea in order to compute the physical module identities involved in a package before typechecking it. However, Backpack distinguishes the two passes—shaping and typing—using completely separate judgments and rules. Although this leads to a doubling of rules, the rules themselves are (we feel) much easier to understand. In particular, the account of linking given by the sequencing rules SHSEQ and TYSEQ is considerably simpler than MixML's formidable linking rule. Moreover, Backpack stages the shaping pass over a whole package entirely before the typing pass, leading to a clearer conceptual split between the two phases of package typechecking than in MixML, where the two passes are interleaved.

A key reason we can get by with a simpler semantics of linking is that we are deliberately less ambitious than MixML in a certain sense: unlike MixML, we do not aim to completely subsume the

functionality of ML modules. MixML does, and this means that its semantics must deal with nested uses of *translucent sealing* (*i.e.*, the ability to define types that are "transparent" inside a module

but "opaque" outside), a defining feature of ML modules which compounds the already-tricky double-vision problem. In contrast, Backpack does not attempt to support translucent sealing—and thus does not suffer the attendant complexities—for the simple reason that Haskell, our target of elaboration, cannot support it.

Finally, MixML is defined by elaboration into an internal language, LTG, which was designed specifically to capture all the necessary features of MixML. (LTG is an extension, with linear kinds, of an earlier IL, similarly specialized for recursive ML module systems, called RTG [9].) LTG's unconventional metatheory underscores MixML's status as a foundational calculus rather than a practical language design, in contrast to Backpack, whose IL is a formalization of an existing implementation artifact, the GHC module system. A major benefit of our approach is that the semantics (via elaboration) of a Backpack package may be understood by Haskell programmers essentially in terms of a reshuffling of import and export lists in their Haskell modules. The elaboration in Figure 10 is a case in point.

Logical Module Names vs. Physical Module Identities Module identities, which establish canonical *physical* names for modules (as distinct from program-level *logical* names), serve two important roles in Backpack's semantics: (1) they simplify and regularize the elaboration into Haskell modules (and its soundness proof), and (2)

they are the principal component of our solution for how to support applicative mixin linking.

Concerning the first point: The distinction between logical and physical names is a central technical element enabling—and conceptually reinforcing—the elaboration into our Haskell-based IL. In particular, a key invariant of elaboration is that the physical part of a package's EL type gives a precise description of the IL modules that it elaborates to; the logical part of its type is only relevant for namespace management during Backpack-level typechecking.

Concerning the second point: The idea of distinguishing between logical and physical names is not new. A number of prior formalisms for ML-style modules—including the Definition of Standard ML itself—rely on a similar distinction [22, 29, 28]. The key advantage of this approach (as opposed to more direct, syntactic type systems for modules, *e.g.*, [20, 15]) is that physical identities greatly simplify the treatment of type equality in the presence of aliasing: no matter their logical names, two types are equal iff they have the same physical identity. This eliminates the need for fancier mechanisms for handling type sharing, like translucent sums or singleton kinds (see Rossberg *et al.* [28] for further discussion). Moreover, for recursive and mixin module extensions of ML [8, 27], the logical/physical distinction has enabled clean solutions to the double vision problem, as discussed above. (There is some recent work by Im *et al.* [19] on solving double vision "syntactically"—*i.e.*, using only logical names—but it does not account for separate typechecking of mutually recursive modules in general.)

What distinguishes Backpack from these prior systems is its support for *both* separate typechecking of recursive modules *and* an applicative semantics of instantiation, as appropriate for a pure language like Haskell. To handle the combination, we needed to

enrich the language of module identities with both (equi-)recursive μ-binders and constructor applications, and employ (standard) unification and equivalence-checking algorithms that work for these recursive identities [18, 14]. To see why, consider the example from Section 2.6, in which the modules A and B in package ab-rec-sep have the recursive identities ν_A and ν_B defined on the subsequent page. If one were to define another package ab-rec-sep2 in the same way, the identities of A and B would be exactly the same. In contrast, were we to code up this example in MixML, each distinct package defined like ab-rec-sep would produce modules with "fresh" (distinct) identities, as one would expect given MixML's *generative* semantics of instantiation. Nevertheless, we observe that

recursive identities do not complicate the semantics much, a testament to the scalability of the logical/physical approach.

Separate Compilation for ML Setting aside the lack of support for recursive linking, ML functors are not by themselves really a practical mechanism for SMD due to the proliferation of "sharing" constraints that are known to arise when programming in a "fully functorized" style (*i.e.*, in which modules are parameterized explicitly, via the functor mechanism, over all their dependencies). Consequently, a number of systems have been proposed for building a better SMD framework on top of the existing ML module system.

Before discussing these systems in more detail, let us observe two important ways in which they all differ from Backpack. First, unlike Backpack, the separate compilation systems for ML build improved SMD support on top of the already-powerful ML module system, which offers instantiation, reuse, and at least some form of SMD via functors. In contrast, Backpack is built on top of Haskell, which lacks those features, and thus the expressiveness boost it offers over the underlying language is in some sense more significant. Second, we realize this boost not through functors but through mixins; as a result, Backpack supports recursive linking, and avoids the need for any separate notion of sharing constraints.

To address the lack of separate compilation in ML, Cardelli introduced a foundational calculus of *linksets* [5]. However, as a foundational calculus, this framework lacks support for user-defined abstract data types, as well as recursion at the module or core level—two prominent features that drive the complexity of state-of-the-art module systems. Building on Cardelli's linkset foundation, Swasey *et al.* [31] designed a typed language of program fragments, SMLSC, that organizes lists of top-level SML definitions (*e.g.*, modules) into what they call *units*. Linking happens automatically by name when unit definitions are considered in the

same linkset. In particular, when multiple units in a linkset have "interface imports" on some common name, those dependencies unify automatically without extra annotations. SMLSC units therefore eliminate the need for sharing constraints on dependencies—as mixin modules do—but they do not permit recursive linking.

In a different vein, targeting "open" modular programming, the Acute language of Sewell *et al.* [30] and the Alice ML language of Rossberg *et al.* [26] support not only separate compilation, but dynamic linking, marshalling/pickling, and (in the case of Acute) versioning of components, all of which are beyond the scope of Backpack. While Acute repurposes modules (with new primitive operations) as a mechanism for compilation units and linking, Alice ML defines "components" by reduction to a simpler construct of "packages" (modules as first-class core values). Linking in Acute consists of (non-recursive) chains of module definitions and imports, whereas Alice ML employs a more flexible and dynamic "component manager" approach based on Java class-loading (rather than linksets). Neither Acute nor Alice ML supports recursive modules.

As part of the OCaml module system [21], the `ocamlc` compilation tool performs separate compilation on files that contain module components. The tool treats the file system rather like a mixin: each component (*i.e.*, a file) can be defined as an implementation (*i.e.*, a `.ml` file) or a hole (*i.e.*, a `.mli` file), and components can be recursively linked. Like SMLSC but unlike Backpack, though, these "mixins" cannot be instantiated and reused: a separately-compiled file cannot be linked with multiple implementations of its dependencies. In essence, `ocamlc` implements something similar to the target IL of Backpack's elaboration, albeit for OCaml (obviously) and extended with full separate compilation rather than just separate typechecking. It does not, however, provide a *language* for building and linking components, as Backpack does.

Mixin Modules for OO Languages Our focus has been on mixin-based SMD in the setting of a typed *functional* language. In the

object-oriented community, mixins have already seen significant uptake. Both Scala [24] and J& [23], for instance, incorporate mixin-style composition into the very fabric of their designs. However, as we have explained, we are particularly interested in the question of how to retrofit *existing* languages with mixin-based SMD, and to our knowledge there is relatively little work on that.

The SMARTJAVAMOD/*component* systems of Ancona *et al.* [2] define a new level of mixin modules to encapsulate existing Java classes. A component contains *defined* classes and *deferred* classes, the latter of which are specified as abstract class names with various constraints. The "bind" construct, which performs mixin linking, instantiates components generatively, producing a unique copy of all the classes inside the merged result (and thus fresh abstract types). In contrast, Backpack supports an applicative semantics of instantiation.

The SMARTJAVAMOD/component languages are implemented with a translation into "polymorphic bytecode" [1], essentially an extension of JVM bytecode with markers and constraints for the deferred classes (*i.e.*, holes) in the component. For that reason, their IL resembles Backpack's, although they present no formal definition of their elaboration into this language. Instead, they define a reduction semantics on components that flattens them into fully instantiated Java class definitions. In Backpack, following much work on ML module systems [15, 28, 27], packages do not have a direct reduction semantics—rather, their meaning is given by a formal

translation into a typed IL, in our case based on Haskell.

As is the tradition in object-oriented languages, the aforementioned systems emphasize dynamic binding, virtual methods, overriding, etc., and do not consider the issue of the double vision problem. In contrast, Backpack supports only static binding, does not permit overriding, and invests great effort to avoid double vision.

5. Future Work

Type Classes Backpack modules only allow data types and values; type classes and type class instances are conspicuously absent. We have left them out of the system deliberately in order to focus attention on the essential features of Backpack that we hope will be broadly applicable, not just to Haskell. That being said, we believe that incorporating type classes into Backpack should be feasible.

In the extension we envision, type classes and instances would both be new kinds of core entities, although the latter would differ from existing entities in that (1) they do not have simple syntactic names and (2) import resolution treats them differently (see below). As with all entities, the export specs ($espc$) listed in a module type would denote which instances a module provides to its clients.

The interaction between instances and signatures poses an interesting challenge: linking an implementation for a hole, or even linking two holes together through aliasing, might result in the existence of two distinct instances for the same type class and type that are visible within a single module. For example, in the bindings

```
P = [class Eq a where ...]
A :: [data T]
```

$$
\begin{array}{lll}
\mathsf{B} & :: & [\texttt{data T}] \\
\mathsf{C} & = & \left[\begin{array}{l}
\texttt{import P} \\
\texttt{import qualified A} \\
\texttt{import qualified B} \\
\texttt{instance Eq A.T where ...} \\
\texttt{instance Eq B.T where ...}
\end{array}\right]
\end{array}
$$

well-typedness of C requires that A.T and B.T be distinct types. But if we then add an alias binding A = B, these two types are unified into a single one, and now C defines two different instances for a single class and type, making it ill-typed.

To prevent this form of error, we would need to amend the definition of merging for sets of export specs ($\overline{espc}$) to prevent two distinct instances for the same class and type from merging together successfully. This would be enough to guarantee that the substitution that links A and B together would not be well defined on the package type for the previous four bindings, thus rejecting the addition of the alias binding A = B.

We would also need to extend the IL in order to maintain a crucial invariant of the elaboration translation, namely that the IL translation of a module only imports the entities that were visible to that module during Backpack-level typechecking (Section 3.4). Naively, this invariant would not be preserved in the presence of type classes because Haskell does not support named instances, so there is no way to explicitly delimit the instances that one module imports from another. For example, suppose a module Y imports a hole X, and then the hole is subsequently filled with an implementation defining an instance that was not in the signature for X. In that case, we do not want the elaboration of Y to suddenly see that new instance, since it might break Y's well-typedness, but there is no way *a priori* to prevent it.

To restore this invariant in the presence of type classes, we would need to introduce—*only in the IL*—the ability to explicitly name instances on import statements. This capability, which would require a minor extension to GHC, would then allow us, when elaborating Y in the above example, to explicitly restrict the import of X to only include those instances that were visible in X's signature.

Type Synonyms and `newtype` We could straightforwardly extend Backpack with both type synonyms and Haskell's `newtype` mechanism for defining abstract data types. Both would be separate entities along with datatypes and values, with accompanying defined entity specs (*dspc*); because they are core entities, they would be imported, exported, and recorded in module types just like datatypes and values. However, for compatibility with GHC, neither would be declarable "abstractly" in signatures (*i.e.*, by omitting the "right-hand sides"), unlike regular data types.

In the case of type synonyms, we would need to treat them as transparently equal to their defining types. To accomplish this, we would simply expand type synonyms as part of Backpack elaboration, ensuring that they never appeared in our semantic objects (the "F-ing modules" approach of Rossberg *et al.* [28] works similarly). Since the synonyms themselves would never appear in any semantic types (*typ*), they would not complicate type equality. In contrast, `newtypes` would not be automatically equated with their defining types; in semantic types (*typ*) they would look and behave essentially like regular data types.

Versioning Lastly, while the support for versioning in Cabal (Haskell's existing package management system) does not obviate interfaces and mixins, neither do interfaces and mixins obviate versioning. An important direction for future work is to investigate how best to integrate versioning into Backpack.

Acknowledgments

We are grateful for innumerable impromptu whiteboard discussions with Neel Krishnaswami, Joshua Dunfield, Aaron Turon, Beta Ziliani, and Georg Neis; for early design discussions with Claudio Russo, Dimitrios Vytiniotis, and Duncan Coutts; and for detailed technical feedback from Andreas Rossberg.

References

[1] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: compositional compilation for Java-like languages. In *POPL '05*.

[2] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Flexible type-safe linking of components for Java-like languages. In *JMLC '06*.

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

[4] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *ICCL '92*.

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

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

[7] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A formal specification of the Haskell 98 module system. In *Haskell '02*.

[8] Derek Dreyer. A type system for recursive modules. In *ICFP '07*.

[9] Derek Dreyer. Recursive type generativity. *JFP*, 17(4&5), 2007.

[10] Dominic Duggan. Type-safe linking with recursive DLLs and shared libraries. *ACM TOPLAS*, 24(6):711–804, 2002.

[11] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *ICFP '96*.

[12] Karl-Filip Faxén. A static semantics for Haskell. *JFP*, 12(5), 2002.

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

[14] Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *ICFP '04*.

[15] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[16] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. *ACM TOPLAS*, 27(5):857–881, 2005.

[17] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *HOPL III*, 2007.

[18] Gérard Huet. *Résolution d'équations dans des langages d'ordre* 1, 2, . . ., ω. PhD thesis, Université Paris 7, September 1976.

[19] Hyeonseung Im, Keiko Nakata, Jacques Garrigue, and Sungwoo Park. A syntactic type system for recursive modules. In *OOPSLA '11*.

[20] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95*.

[21] Xavier Leroy. The Objective Caml system: Documentation and user's manual. `http://caml.inria.fr/ocaml/htmlman/`.

[22] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The*

Definition of Standard ML (Revised). MIT Press, 1997.

[23] Nathaniel Nystrom, Xin Qi, and Andrew Myers. J&: Nested intersection for scalable software composition. In *OOPSLA '06*.

[24] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05*.

[25] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *ICFP '06*.

[26] Andreas Rossberg. The missing link – dynamic components for ML. In *ICFP '06*.

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

[28] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *TLDI '10*. Extended version available from the author's website at: `http://www.mpi-sws.org/~rossberg/f-ing`.

[29] Claudio V. Russo. *Types for Modules*. PhD thesis, University of Edinburgh, 1998.

[30] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. *JFP*, 17(4–5), 2007.

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