From Structures and Functors to Modules and Units

Scott Owens Matthew Flatt
University of Utah
{sowens,mflatt}@cs.utah.edu

Abstract

Component programming techniques encourage abstraction and reuse through external linking. Some parts of a program, however, must use concrete, internally specified references, so a pure component system is not a sufficient mechanism for structuring programs. We present the combination of a static, internally-linked module system and a purely abstractive component system. The latter extends our previous model of typed units to properly account for translucency and sharing. We also show how units and modules can express an SML-style system of structures and functors, and we explore the consequences for recursive structures and functors.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Design, Languages

Keywords Module, component, structure, functor, unit

1. Introduction

A complete module system for a typed functional language must support a wide range of features for managing types and organizing programs, including generativity, translucency, and the sharing of types, as well as nesting, recursive dependency, and flexible composition of the modules themselves. In particular, module composition should support two distinct linking mechanisms: internal linking and external linking.

Internal linking supports definite reference to a specific implementation of an interface. Internally linked module systems are common; examples include Java's packages and classes, ML's structures, and Haskell's modules. In each of these systems, two modules are linked when one directly mentions the name of another, either with a dotted path (`ModuleName.x`), or import statement (`import ModuleName`).

External linking supports parameterized reference to an arbitrary implementation of an interface. ML's module system includes a `functor` construct that supports external linking. A functor declares an interface over which its definitions are parameterized; the parameterization is resolved outside of the functor with a functor application, in analogy to function application. A functor application supplies a structure that satisfies the functor's declared interface, and it produces another structure. Thus, the `functor` and `structure` constructs are tied closely together by the functor application linking mechanism.

Unlike functions, functors cannot be recursively defined, which keeps them from expressing patterns where mutually dependent program features are split across component [30] boundaries. We argue that this results from the close coupling of the internal and external linking mechanisms. Each structure produced by a functor application is potentially accessible with a direct link, which means that its contents must be fully defined, even if it is not intended to be the final result of a sequence of applications.

This paper presents a module system based on our experience developing PLT Scheme. The system supports both internal linking and recursive external linking, and it uses a different construct for each. We call the structure-like construct **module**, and the functor-like component construct **unit**. The coupling between modules and units is looser than between structures and functors; external linkages between units can be resolved incrementally without producing intervening modules. Loose coupling is crucial in allowing units to naturally support compositions that contain cyclic linkages. It also permits us to address compilation in the module layer, and place compilation management functionality there without complicating the component layer.

The `unit` construct extends our previous model of units [13] to support cooperation with modules, and to handle translucent and opaque type imports and exports. We explain our `module`-`unit` separation, and relate its expressive power to that of the SML module system. In the process, we offer an alternate semantics for the SML module system through compilation into `module` and `unit`. For SML programmers, our alternate view may offer insights into enabling mutual recursion among SML functors. For non-SML programmers, our alternate semantics hopefully complements the existing semantics of SML and of units to foster a deeper understanding of key module and component system technologies.

Section 2 introduces units and modules and shows how they complement each other. Section 3 contains several examples of the expressiveness of units. Section 4 relates units and modules to functors and structures, and gives a translation from a model of generative structures and functors into modules and units. Section 5 discusses necessary extensions to our model for practical programming . The type system for units (section 6) adapts ideas from type systems for SML modules, in particular manifest types [20], to support sharing and translucency. Section 7 presents a reduction semantics and soundness theorem.

2. Modules and Units

Figures 1, 2, and 3 present the syntax of our language of modules and units over a simply-typed core language that includes sum and product types, value definitions, and type definitions. The module system comprises the **module** and path constructs, and the unit system comprises the **unit** and **compound** expressions and the **invoke** definition.

ICFP'06 September 16–21, 2006, Portland, Oregon, USA.

m, t, x	=	module, type, and value names			
name	=	$m \mid t \mid x$	P_m	=	$m^i \mid P_m.m \mid P_m.\widehat{m}$
ident	=	$m^i \mid t^i \mid x^i$	P_x	=	$x^i \mid P_m.x \mid P_m.\widehat{x}$
i is an infinite set of stamps			P_t	=	$t^i \mid P_m.t \mid P_m.\widehat{t}$

Figure 1. Names, identifiers, and paths

Ty	=	**int** $\mid$ *Ty*+*Ty* $\mid$ *Ty* $\times$ *Ty* $\mid$ *Ty* $\rightarrow$ *Ty* $\mid P_t$	*K*	=	$\star$
	$\mid$	**unitT** Γ (*name** $\rightarrow$ *name**)	Γ	=	$\epsilon \mid \Gamma$,*B*
B	=	x^i:*Ty* $\mid$ t^i:*K* $\mid$ t^i:*K*=*Ty* $\mid$ m^i:$\mathcal{M}$			
$\mathcal{M}$	=	Γ **provide** *name**			

Figure 2. Type grammar

Prog	=	M^*
M	=	**module** m^i **provide** *name** = *Ds*
	$\mid$	**module** m^i:Γ **provide** *name** = *Ds*
Ds	=	$\epsilon \mid D\ Ds$
D	=	x^i:*Ty*=*E* $\mid$ t^i:*K*=*Ty* $\mid$ **invoke** *E* **as** m^i:Γ $\mid$ *M*
E	=	$\mathbb{Z} \mid P_X \mid$ **injl** *E* $\mid$ **injr** *E* $\mid$ (*E*,*E*) $\mid \pi_1 E \mid \pi_2 E$
	$\mid$	**case** *E* **of** *E* + *E* $\mid \lambda x^i$:*Ty*.*E* $\mid$ *E E*
	$\mid$	**unit import** Γ **export** Γ. *Ds*
	$\mid$	**compound import** Γ **export** Γ **link** U^* **where** L^*
U	=	*E*:**import** Γ **export** Γ
L	=	*ident* $\leftarrow$ *ident*

Figure 3. Term grammar

We present the language without inference, so each value and type definition comes with an annotation of its type (*Ty*) or kind (*K*), respectively. The expressions (*E*) of the language are typical of a simply-typed λ-calculus, with the addition of the unit system. Thus, the language has only one kind, and it has types for integer, sum, product, function, and unit values. A type can also contain a path to a type definition.

2.1 Modules

A program (*Prog*) is a sequence of modules (*M*), each of which contains a sequence (*Ds*) of definitions (*D*). Each definition's scope extends from the immediately following definition to the end of the module. Definitions are accessed from outside of the module with a path to either a value (P_X) or to a type (P_t). For example, the path $m^1.n.t$ refers to the definition of type t inside of module n which is itself defined inside of m^1, a module defined in the scope where the path appears.

We follow Leroy [20] and use identifiers (x^i, t^i, m^i) for references within the module, and names (x, t, m) for all references from outside of the module. Our α-relation for module-level definitions is permitted to modify only the stamp on a name, and cannot change the name itself. This invariant lets us use names to access module bindings and to freely α-rename identifiers without interference.

A path can contain both names and hatted names ($\widehat{x}$, $\widehat{t}$, $\widehat{m}$). A name-based reference into a module is valid only when that name is listed in the module's **provide** clause, but a hatted-name reference ignores the **provide** clause to allow access to unprovided module bindings. Thus, hatted names are prohibited in source programs; however, the path grammar and type system support them because the type system can require access to unprovided type definitions during type checking (see section 6.1).

The need for hatted names arises because the **module** construct does not support sealing, so the full types of values extracted from the module are visible from outside of the module. This design decision makes our module construct more like Haskell's `module` than ML's `structure`, and it allows our model to support sealing functionality in a single place: units.

As our form for managing internal linking, modules have two roles. First, module definitions can be nested for fine-grained management of name grouping and lexical scoping. Second, top-level modules naturally form the boundaries of *separate compilation*, because every dependency on another top-level module is explicit in the body of the module. To facilitate separate compilation for modules, we restrict a program's top-level to module definitions only and require that the dependency relation between modules be acyclic.

2.2 Units

Component programming emphasizes independent development and deployment of components, as well as component re-use. Units support independent deployment with external linking, which lets each of a component's dependencies be satisfied at its point of deployment, instead of at its point of implementation. Recursive linking forms an important part of independent deployment, because any two units can be linked, as long as they satisfy the appropriate interfaces.

Independent deployment also requires *independent* or *compositional* compilation of components (not merely *separate* compilation as for modules). Each of a component's indirect references must have a signature that describes the compile-time information for the reference, so that the unit can be compiled without any knowledge of other units that might be used to satisfy the import. Since all of a unit's imported values are supplied externally, and their types are part of the compile-time signature, units are naturally first-class values, similar to functions. Unit linking is an expression, similar to function application; consequently, units support dynamic, run-time linking.

A **unit** expression creates an atomic unit, with explicit imports and exports described by contexts (Γ). The unit's body must define each exported binding (*B*), and its body can refer to any imported binding. Each imported or exported binding can specify a value, opaque type, translucent type, or module. Module bindings support the grouping of imported and exported values, types, and modules. The type of a unit includes a single context that contains the unit's imported and exported bindings, and it also includes a listing of which of these bindings are imports and which are exports. The single context can interleave imported and exported bindings, allowing the types of imported values to refer to types exported from the unit. This feature helps avoid the "double-vision" problem, as we show in section 4.3.

Units use identifiers and names to distinguish internal and external names, similar to modules. Thus, the α-relation for units renames imported and exported definitions inside of the unit by altering only stamps. However, definitions that are not imported or exported have no need for external consistency, so their names and stamps both can be changed; this flexibility is needed for the operational semantics.

A **compound** expression links units together to form a new unit. Each linked unit (*U*) specifies an expression along with the imports and exports expected of the expression. The linkages (*L*) specify which type, value, and module exports from the linked units are used to satisfy each of the linked units' imports. Each identifier on the left side of a linkage must be listed in one of the import contexts accompanying the linked units, and each identifier on the right of a linkage must be listed in one of the export contexts, or in the compound unit's imports. The compound's **export** clause specifies which of the linked units' exports are themselves exported from the compound unit. An **invoke** expression evaluates the definitions inside of a unit that has no imports, and it places the exported values and types in the enclosing scope, accessible through the given module identifier.

```
module ordered_int provide oi_unit =
 oi_unit:unitT ... =
  unit import [] export [item: ⋆ =int, compare:item → item → bool].
   item: ⋆ =int
   compare:int → int → bool = ...

module set provide set_unit t =
 t: ⋆ =
  unitT
   [itm: ⋆ , set: ⋆ , cmp:itm → itm → bool, insert:itm → set → set]
   (cmp itm → insert set)
 set_unit:t =
  unit import [itm: ⋆ , cmp:itm → itm → bool]
       export [set: ⋆ , insert:itm → set → set].
   set: ⋆ = ... itm ...
   insert:itm → set → set = ... cmp ...

module main provide s =
 t: ⋆ = unitT [set: ⋆ , item: ⋆ =int, insert:item → set → set]
              ( → insert item set)
 int_set_unit : t =
  compound import []
              export [set: ⋆ , item: ⋆ =int, insert:item → set → set]
   link ordered_int.oi_unit : import ... export ...
        set.set_unit : import ... export ...
   where (cmp ← compare) (itm ← item)
 invoke int_set_unit as set:[set: ⋆ , item: ⋆ =int, insert:item → set → set]
 s:set = set.insert 12 ...
```

Figure 4. Set example

```
module math provide add add_tuple=
 add:int → int → int= ...
 add_tuple:int × int → int= ...

module o_mod provide o_unit =
 o_unit : unitT [a: ⋆ ,b: ⋆ ,c: ⋆ ,o:(b → c) → (a → b) → a → c]
                (a b c → o) =
  unit import [a: ⋆ ,b: ⋆ ,c: ⋆ ] export [o:(b → c) → (a → b) → a → c].
   o : (b → c) → (a → b) → a → c =
    λ f:b → c. λ g:a → b. λ x:a. f (g x)

module t_mod provide types_unit =
 types_unit : unitT [t: ⋆ =int × int,u: ⋆ =int,v:int → int] ( → t u v) =
  unit import [] export [ ... ].
   t: ⋆ =int × int   u: ⋆ =int   v: ⋆ =int → int

module c_mod provide c_unit =
 c_unit : unitT ... =
  compound import []
              export [o:(u → v) → (t → u) → t → v,
                      t: ⋆ =int × int,u:=int,v: ⋆ =int → int]
   link o_mod.o_unit:import ... export ...
        t_mod.types_unit:import ... export ...
   where (a ← t) (b ← u) (c ← v)

module ex provide ex6 =
 invoke c_mod.c_unit as o: ...
 ex6:int = (o.o math.add math.add_tuple) (1, 2) 3
```

Figure 5. Polymorphism

A subtype relation $<:$ arises naturally from unit values. Roughly speaking, a unit with less imports or more exports can be used in place of a unit with more imports and less exports. Section 6.2 discusses subtyping in further detail.[1]

2.3 Example

Figure 4 presents the skeleton of a set library that is parameterized over the items in the set. Identifier stamps are omitted for clarity, but the requisite type annotations and full unit import and export contexts are present to illustrate each facet of the example. A practical language that incorporates units would avoid this verbosity with type inference and a signature facility for abstracting out common import/export patterns (see section 5).

The *ordered_int* module provides a unit *oi_unit* that exports a type, *item*, and a value, *compare*, that compares two items. The unit's body defines *item* to be **int**, and the unit exports it translucently, so that *item* is known to be **int** when *oi_unit* is used.

The module *set* provides a unit, *set_unit*, that imports a type, *itm*, and a value, *cmp*, and that exports a type *set* and a value *insert*. The body of *set_unit* references its imports in the definition of its exports. Because *itm* is imported as an opaque type, *set_unit*'s body has no information about it, other than its existence.

Although the implementation of the insert function in *set_unit* is elided, it would include many direct references to standard library functions managed by the module system. If the programmer wishes to support multiple implementations of these library functions, they would become unit imports instead. In this way, modules manage direct references, whereas units manage indirect references.

The *main* module links *oi_unit* and *set_unit* into *int_set_unit*, and it invokes the result to allow direct access to the insert function for sets of integers. The compound unit *int_set_unit* exports the type *set* opaquely, but exports the type *item* translucently, so that the *insert* function can add integers to the set without exposing the set's implementation details. The **invoke** statement runs *int_set_unit* and groups its exports under the module name *set*.

[1] The complications of subtyping in the core language can be avoided by taking units out of the core and placing them in a module level, similar to ML's functors.

3. Expressiveness

Units can express several basic language constructs, including parametric polymorphism, recursive functions, and recursive datatypes. Along with the set example (figure 4), these examples illustrate the basic methods that units use to support recursive linking and type abstraction. Section 4.2 shows that they can also express structures and functors.

A polymorphic function can be encoded as a unit that imports the relevant type variables and exports the function. Figure 5 contains such a definition of the polymorphic function composition combinator, *o*. The unit *o_unit* defines the function, and the units *types_unit* and *c* instantiate it to particular types. Because *o_unit* can be independently compiled, the implementation of *o* is not duplicated when the compound unit *c* is created or invoked. Furthermore, since *o_unit* is a first-class value, it represents *o* as a first-class polymorphic function.

To implement a recursive function, a unit imports the function to be used for recursive calls, and links with itself. Figure 6 defines a factorial function with this technique. The unit *fact_unit* defines and exports the function *fact*, which relies on the imported *facti* function for recursive calls. Linking the unit by itself, providing the exported *fact* function for the imported *facti*, creates the function.

The same technique works at the type level to support recursive datatype definitions, such as lists of integers (figure 7). The unit defines type *list* in terms of an opaquely imported type *listi*, and defines the standard constructors and destructors for lists with reference to *list* and *listi* as needed. The compound unit supplies *list* for the *listi* import, causing each of the exported function's type references to *listi* to become references to *list*. Further, it supplies *u*, which equals **int** for the *list_unit*'s type import. Since *list* is exported opaquely, the fact that *list* depends on *listi* is not known at the site of the compound.

```
module m provide f4 =
 fact_unit : unitT ... =
  unit import [facti:int → int] export [fact:int → int].
    fact:int → int = λ i:int. if0 i then 1 else i × facti (i - 1)
 fact_compound : unitT [fact:int → int] ( → fact) =
  compound import [] export [fact:int → int]
    link fact_unit:import ... export ...
    where facti ← fact
 invoke fact_compound as F:[fact:int → int]
 f4:int = F.fact 4
```

Figure 6. Recursive factorial function

```
module m provide ex l =
 list_unit : unitT [t: ⋆ ,listi: ⋆ ,list: ⋆ ,cons:t → listi → list, ... ]
                   (t listi → list cons ...) =
  unit import [t: ⋆ ,listi: ⋆ ] export [list: ⋆ ,cons:t → listi → list, ... ].
    list: ⋆ =int+(t × listi)
    cons:t → listi → list = λ x:t. λ l:listi. injr (x, l)
    nil:list = injl 0
    hd:list → t = λ l:list. case l of (λ n:int. ...) + (λ c:t × listi. π1 c)
    tl:list → listi = λ l:list. case l of (λ n:int. ...) + (λ c:t × listi. π2 c)
 int_u: ... = unit import [] export [u: ⋆ =int]. u: ⋆ =int
 list_compound : unitT ... =
  compound import [] export [list: ⋆ ,cons:int → list → list, ... ]
    link list_u:import ... export ...
         int_unit:import ... export ...
    where (listi ← list) (t ← u)
 invoke list_unit as l: ...
 ex:int = l.hd (l.cons 1 l.nil)
```

Figure 7. Recursive datatype example

4. Structures and Functors

Units and modules have close counterparts in the structures and functors of Standard ML. A structure is a first-order entity that groups value, type, and structure definitions under a single name. The biggest difference between structures and modules is that a structure can be *sealed* with a signature, hiding the definitions of certain specified types and values. In contrast, our module construct can hide names from direct access, but it does not create abstract types by hiding type definitions.

A functor is a function that consumes and produces structures, including, of course, the types and values contained therein. Similarly, a unit consumes and produces values, types, and modules; furthermore, types can be imported or exported opaquely, capturing the sealing functionality of structures. However, functor application not only uses the argument structure to satisfy the functor's imports, but it also immediately produces the result structure. The unit system's separation between linking and invocation (i.e. unit linking produces a unit, and unit invocation produces a module) is a crucial difference between units and functors.

A higher-order functor can import and export other functors in addition to structures. Higher-order functors support incremental linking patterns, such as currying to simulate multiple-argument functors. Units naturally support incremental linking, and a unit can import or export another unit value if desired.

Figure 8 defines a language of structures and first-order, generative functors with a type system (figure 9) based on manifest types [20]. (We adopt notation from our type system for units; see figure 15.) The notable departures from SML include the ability to freely nest structures and functors, the requirement that each structure definition has an explicitly given context, and the requirement that the functor application construct have context and functor type (F') annotations (the second and third are discussed in section 4.2).

$$
\begin{array}{lcl}
m & = & \text{structure or functor name} \\
Ty' & = & \mathbf{int} \mid Ty' \mathbf{+} Ty' \mid Ty' \times Ty' \mid Ty' \to Ty' \mid P_t \\
B' & = & x^i\mathbf{:}Ty' \mid t^i\mathbf{:}K \mid t^i\mathbf{:}K\mathbf{=}Ty' \mid m^i\mathbf{:}\Gamma' \mid m^i\mathbf{:}F' \\
\Gamma' & = & \epsilon \mid \Gamma'\mathbf{,}B' \\
F' & = & (m^i\mathbf{:}\Gamma' \to \Gamma') \\
D' & = & x^i\mathbf{:}Ty'\mathbf{=}E' \mid t^i\mathbf{:}K\mathbf{=}Ty' \mid \mathbf{structure}\ m^i\mathbf{:>}\Gamma'\mathbf{=}D'^{*} \\
 & \mid & \mathbf{structure}\ m^i\mathbf{:>}\Gamma'\mathbf{=}P'_m\mathbf{:}F'(P'_m\mathbf{:}\Gamma') \mid \mathbf{functor}\ m^i\mathbf{:}F'\mathbf{=}D'^{*} \\
E' & = & \mathbb{Z} \mid P_X \mid \mathbf{injl}\ E' \mid \mathbf{injr}\ E' \mid (E'\mathbf{,}E') \mid \pi_1 E' \mid \pi_2 E' \\
 & \mid & \lambda x^i\mathbf{:}Ty'.E' \mid E'E' \mid \mathbf{case}\ E'\ \mathbf{of}\ E'\mathbf{+}E'
\end{array}
$$

Figure 8. Language with structures and functors

$\Gamma' \vdash \Gamma' \qquad \Gamma' \vdash F' \qquad \Gamma' \vdash Ty' : K$
ensure that a structure type (represented as a context), a functor type, or a value type are well-formed

$\Gamma' \vdash \Gamma' <: \Gamma' \qquad \Gamma' \vdash F' <: F' \qquad \Gamma' \vdash Ty' \equiv Ty'$
subtyping for structure and functor types, and type equality for value types

$\Gamma' \vdash E' : Ty'$
typing for expressions

$$\boxed{\Gamma' \vdash D'^{*} : \Gamma'}$$

$$\frac{}{\Gamma \vdash \epsilon : \epsilon}\ \textsc{Ds}\epsilon' \qquad \frac{\Gamma_1 \vdash D : B \qquad \Gamma_1\mathbf{,}B \vdash Ds : \Gamma_2 \qquad \sigma(B) \notin \textsc{dom}(\Gamma_1)}{\Gamma_1 \vdash D\ Ds : B\mathbf{,}\Gamma_2}\ \textsc{Ds}'$$

$$\boxed{\Gamma' \vdash D' : B'}$$

$$\frac{\Gamma_1 \vdash E : Ty_2 \qquad \Gamma_1 \vdash Ty_1 : \star \qquad \Gamma_1 \vdash Ty_2 \equiv Ty_1}{\Gamma_1 \vdash x^i\mathbf{:}Ty_1\mathbf{=}E : x^i\mathbf{:}Ty_1}\ \textsc{Dval}'$$

$$\frac{\Gamma_1 \vdash Ty : K}{\Gamma_1 \vdash t^i\mathbf{:}K\mathbf{=}Ty : t^i\mathbf{:}K\mathbf{=}Ty}\ \textsc{Dtype}'$$

$$\frac{\Gamma_1 \vdash Ds_1 : \Gamma_3 \qquad \textsc{distinct}_\rho(\Gamma_3) \qquad \Gamma_1 \vdash \Gamma_2 \qquad \Gamma_1 \vdash \Gamma_3 <: \Gamma_2}{\Gamma_1 \vdash \mathbf{structure}\ m^i\mathbf{:>}\ \Gamma_2 \mathbf{=} Ds_1 : m^i\mathbf{:}\Gamma_2}\ \textsc{Dstr}'$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \Gamma_2 \qquad \Gamma_1 \vdash \Gamma_3 \\ \Gamma_1 \vdash F_1 \qquad \Gamma_1 \vdash mP_1 \mapsto m_2^i\mathbf{:}F_2 \qquad \Gamma_1 \vdash mP_2 \mapsto m_3^i\mathbf{:}\Gamma_4 \\ \Gamma_1 \vdash F_2 <: F_1 \qquad \Gamma_1 \vdash \Gamma_4 <: \Gamma_3 \qquad F_1 = (m_4^i\mathbf{:}\Gamma_5 \to \Gamma_6) \\ \Gamma_1 \vdash \Gamma_3 <: \Gamma_5 \qquad \Gamma_1 \vdash \Gamma_6\{m_4^i \leftarrow mP_2\} <: \Gamma_2\end{array}}{\Gamma_1 \vdash \mathbf{structure}\ m_1^i\mathbf{:>}\ \Gamma_2 \mathbf{=} mP_1\mathbf{:}F_1(mP_2\mathbf{:}\Gamma_3) : m_1^i\mathbf{:}\Gamma_2}\ \textsc{Dfapp}'$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \Gamma_2 \qquad m_2^i \notin \textsc{dom}(\Gamma_1) \qquad (m_2^i\mathbf{:}\Gamma_2), \Gamma_1 \vdash \Gamma_3 \\ m_2^i\mathbf{:}\Gamma_2, \Gamma_1 \vdash Ds_1 : \Gamma_4 \qquad \textsc{distinct}_\rho(\Gamma_4) \\ m_2^i\mathbf{:}\Gamma_2, \Gamma_1 \vdash \Gamma_3 <: \Gamma_4 \qquad m_2 \notin \textsc{dom}_\rho(\Gamma_3)\end{array}}{\Gamma_1 \vdash \mathbf{functor}\ m_1^i\mathbf{:}(m_2^i\mathbf{:}\Gamma_2 \to \Gamma_3) \mathbf{=} Ds_1 : m_1^i\mathbf{:}(m_2^i\mathbf{:}\Gamma_2 \to \Gamma_3)}\ \textsc{Dftor}'$$

Figure 9. Type system for structures and functors

4.1 Diamond Imports

Diamond import refers to a linking pattern where two functors are instantiated with the same argument, and their results are subsequently linked together. In the final linking step, the initial argument should be known to be the same for both modules being linked. In SML, sharing constraints are used to solve diamond import problems.

A typical example of a diamond import is a parser, set up as in figure 10 (using multiple argument functors to simplify the example). The *front_end* functor imports the types t and u; however, $P.u$ is declared equal to $L.t$, allowing *front_end*'s body to rely on this equality, but requiring functor's arguments to satisfy it.

```
structure sym :> [t: ⋆ ] = (t: ⋆ = . . .)  . . .
functor lexer : (S:[t: ⋆ ] → [t: ⋆ =S.t,get_tok: . . . t . . . ]) = . . .
functor parser : (S:[u: ⋆ ] → [u: ⋆ =S.u,parser: . . . u . . . ]) = . . .
functor front_end : (L:[t: ⋆ ,get_tok: . . . t . . . ],
                     P:[u: ⋆ =L.t,parser: . . . u . . . ] → . . . ) = . . .
structure l:>[t: ⋆ = sym.t, . . . ]=lexer: . . . (sym: . . . )
structure p:>[u: ⋆ = sym.t, . . . ]=parser: . . . (sym: . . . )
structure  . . . = front_end: . . . (l: . . . ,p: . . . )
```

Figure 10. Diamond import with functors

```
invoke (unit import [] export [sym_t: ⋆ ]. sym_t = . . . ) as sym: . . .
 sym_unit: . . . = unit import [] export [t: ⋆ =sym.sym_t]. t: ⋆ =sym.sym_t
 lexer: . . . = unit import [ti: ⋆ ] export [te: ⋆ =ti,get_tok: . . . te . . . ]. . . .
 parser: . . . = unit import [ui: ⋆ ] export [ue: ⋆ =ui,parser: . . . ue . . . ]. . . .
 front_end: . . . =
  unit import [v: ⋆ ,fget_tok: . . . v . . . ,w: ⋆ =v,fparser: . . . w . . . ]
       export [ . . . ]. . . .
 l: . . . = compound import [] export [te: ⋆ =sym.sym_t,get_tok: . . . te . . . ]
            link lexer:import . . . export . . .
                 sym:import . . . export . . .
            where ti ← t
 p: . . . = compound import [] export [ue: ⋆ =sym.sym_t,parser: . . . ue . . . ]
            link parser:import . . . export . . .
                 sym:import . . . export . . .
            where ui ← t
invoke
 (compound import [] export [ . . . ]
   link l:import . . . export . . .
        p:import . . . export . . .
        front_end:import . . . export . . .
   where (v ← te)  (fget_tok ← get_tok)
         (w ← ue)  (fparser ← parser)) as . . .
```

Figure 11. Diamond import with units

The key to this example is that the type *sym*.*t* is written down at a single place, and the rest of the example systematically refers to it using type equations. The same techniques apply to a corresponding unit-based program (figure 11).

Without translucent type exports, the *sym* unit would have to export *t* opaquely, leading *l* and *p* to export *te* and *ue* opaquely, which would prevent them from linking with *front_end*. The solution in such a system is to link *lexer*, *parser*, *front_end* and *sym* all in the same **compound**, so that the opaquely exported type from *sym* is directly imported into each of the other units. Thus, the present unit system provides an increase in expressiveness over our previous work, which did not support translucency. In fact, units can express not only diamond import patterns, but can also express structures and functors directly.

4.2 Translating Structures and Functors

Figure 12 presents a structural translation from the language of structures and functors to the language of modules and units. A structure definition translates into two definitions, a module definition and a unit definition. The module for a definition is created by invoking a unit whose exports are the same as the structure's. By going through a unit, the structure's body is appropriately sealed—using a module directly would allow the structure's opaque type definitions to escape. However, this unit cannot be used for functor application, because the functor might expose the fact that one of its imports refers to this structure in particular, such as the translucent export of *t* in the *lexer* functor of figure 10.

A separate unit is built by the STRUCTUNIT function (figure 13) for use in functor applications. The ΓTODS function constructs the body of this unit from the export context by constructing, for each binding, a definition that refers to the module. The unit's export context is created by STRENGTHEN which transforms opaque type

$$\boxed{B' \Longrightarrow B^*}$$

$$\frac{}{x^i{:}Ty \Longrightarrow x^i{:}Ty} \qquad \frac{}{t^i{:}K \Longrightarrow t^i{:}K} \qquad \frac{}{t^i{:}K{=}Ty \Longrightarrow t^i{:}K{=}Ty}$$

$$\frac{\Gamma' \Longrightarrow \Gamma_1 \qquad \mathcal{M}_1 = \Gamma \textbf{ provide } \textsc{dom}_\rho(\Gamma_1) \qquad \Gamma_2 = \textsc{strengthen}(m^i, \Gamma_1) \qquad \mathcal{M}_2 = \Gamma_2 \textbf{ provide } \textsc{dom}_\rho(\Gamma_2)}{m^i{:}\Gamma'_1 \Longrightarrow m^i{:}\mathcal{M}_1, m\_\text{unit}^i\textbf{:unitT } \text{export}^i{:}\mathcal{M}_2\ (\epsilon \rightarrow \text{export})}$$

$$\frac{F' \Longrightarrow Ty}{m_1^i{:}F' \Longrightarrow m_1\_\text{unit}^i{:}Ty}$$

$$\boxed{F' \Longrightarrow Ty}$$

$$\frac{\Gamma'_1 \Longrightarrow \Gamma_1 \qquad \Gamma'_2 \Longrightarrow \Gamma_2 \qquad ns = \textsc{dom}_\rho(\Gamma_2) \qquad im = m^i{:}\Gamma_1 \textbf{ provide } \textsc{dom}_\rho(\Gamma_1)}{(m^i{:}\Gamma'_1 \rightarrow \Gamma'_2) \Longrightarrow \textbf{unitT } im, \Gamma_2\ (m \rightarrow ns)}$$

$$\boxed{D' \Longrightarrow D^*}$$

$$\frac{E' \Longrightarrow E}{x^i{:}Ty{=}E' \Longrightarrow x^i{:}Ty{=}E} \qquad \frac{}{t^i{:}K{=}Ty \Longrightarrow t^i{:}K{=}Ty}$$

$$\frac{\Gamma' \Longrightarrow \Gamma \qquad Ds' \Longrightarrow Ds \qquad E = \textbf{unit import } \epsilon \textbf{ export } \Gamma.\ Ds \qquad D = \textsc{structUnit}(m^i, \Gamma)}{\textbf{structure } m^i{:}{>}\Gamma'{=}Ds' \Longrightarrow (\textbf{invoke } E \textbf{ as } m^i{:}(\Gamma))\ D}$$

$$\frac{\begin{array}{c}\Gamma'_1 \Longrightarrow \Gamma_1 \qquad \Gamma'_2 \Longrightarrow \Gamma_2 \qquad F' = (m_2^i{:}\Gamma'_3 \rightarrow \Gamma'_4) \\ \Gamma'_3 \Longrightarrow \Gamma_3 \qquad \Gamma'_4 \Longrightarrow \Gamma_4 \qquad im = m^i{:}\Gamma_3 \textbf{ provide } \textsc{dom}_\rho(\Gamma_3) \\ IE_1 = \textbf{import } [im] \textbf{ export } \Gamma_4 \qquad \mathcal{M} = \Gamma_2 \textbf{ provide } \textsc{dom}_\rho(\Gamma_2) \\ IE_2 = \textbf{import } \epsilon \textbf{ export } [\text{exp\_mod}^i{:}\mathcal{M}] \\ D = \textsc{structUnit}(m_1^i, \Gamma_1) \\ E = \textbf{compound import } \epsilon \textbf{ export } \Gamma_1 \textbf{ link } P_2\_\text{unit}{:}IE_2\ P_1\_\text{unit}{:}IE_1 \\ \textbf{where } m_2^i \leftarrow \text{exp\_mod}^i\end{array}}{\textbf{structure } m_1^i{:}{>}\Gamma'_1{=}P_1{:}F'(P_2{:}\Gamma'_2) \Longrightarrow (\textbf{invoke } E \textbf{ as } m_1^i{:}(\Gamma_1))\ D}$$

$$\frac{F' = (m_2^i{:}\Gamma'_1 \rightarrow \Gamma'_2) \qquad F' \Longrightarrow Ty \qquad \Gamma'_1 \Longrightarrow \Gamma_1 \qquad \Gamma'_2 \Longrightarrow \Gamma_2 \qquad Ds' \Longrightarrow Ds \qquad im = \text{m}_2^i{:}\Gamma_1 \textbf{ provide } \text{dom}_\rho(\Gamma_1)}{\begin{array}{c}\textbf{functor } m_1^i{:}F'{=}Ds' \Longrightarrow \\ m_1\_\text{unit}^i{:}Ty{=}\textbf{unit import } im \textbf{ export } \Gamma_2.\ Ds\end{array}}$$

Figure 12. Translation into units

definitions into translucent ones that refer to the given module. This modification of the contexts corresponds to the notion of strengthening found in manifest types.

The following example demonstrates the translation from a structure to a module and unit.

```
structure m :> [t: ⋆ ,x:t] =
 t: ⋆ =int
 x:t=1
⟹
invoke (unit import [] export [t: ⋆ ,x:t]. t: ⋆ =int x:t=1) as m:([t: ⋆ ,x:t])
 m_unit: . . . =
  unit import [] export [exp_mod:([t: ⋆ =m.t,x:t] provide x t)].
    module exp_mod provide x t =
      t: ⋆ =m.t
      x:t=m.x
```

A functor definition translates into the definition of a unit whose imports and exports correspond to the functor's imports and exports. Because a functor imports a single structure that groups all of its imported values and types, the unit imports a single module that performs the same grouping. A functor application becomes a compound unit that links the argument structure's unit to the func-

$\Gamma\text{TODS} : P_m \times \Gamma \to D^*$ STRENGTHEN : $P_m \times \Gamma \to \Gamma$

$$\boxed{\text{STRUCTUNIT} : m^i \times \Gamma \to D}$$

$$\frac{\begin{array}{c} Ds = \Gamma\text{TODS}(m^i, \Gamma_1) \quad \Gamma_2 = \text{STRENGTHEN}(m^i, \Gamma_1) \\ ex = \text{exp\_mod}^i\text{:}\Gamma_2 \textbf{ provide } \text{DOM}_\rho(\Gamma_2) \\ M = \textbf{module } \text{exp\_mod}^i \textbf{ provide } \text{DOM}_\rho(\Gamma_1) \textbf{ = } Ds \\ Ty = \textbf{unitT } ex \textbf{ (} \to \text{exp\_mod)} \end{array}}{\text{STRUCTUNIT}(m^i, \Gamma_1) = m\_\text{unit}^i\textbf{:}Ty\textbf{=unit import } \epsilon \textbf{ export } ex\textbf{. } M}$$

Figure 13. Building the unit for a structure

tor's import. The structure's unit exports its bindings under the single module named "exp_mod" to facilitate functor application.

functor *f* (*i*:[*t*: $\star$,*x*:*t*] $\to$ [*t*: $\star$ =*i.ty*:*t*]) =
t: $\star$ =*i.t* *y*:*t*=*x*
structure *s* :> [*t*: $\star$ =*m.t*,*y*:*t*] = *f*: . . . (*m*: . . .)
$\Longrightarrow$
f: . . . = **unit import** [*i*:([*t*: $\star$,*x*:*t*] **provide** *x t*)] **export** [*t*: $\star$ =*i.t*,*y*:*t*].
t: $\star$ =*i.t* *y*:*t*=*x*
invoke (compound import [] **export** [*y*:*t*,*t*: $\star$ =*m.t*]
link *f*:**import** . . . **export** . . .
m_unit:**import** . . . **export** . . .
where *i* $\leftarrow$ exp_mod) **as** *s*:(. . .)
s_unit: . . . = . . .

The functor application construct requires that the paths to the functor and structure are annotated with their types. These annotations are used to create the unit types in the **link** section of the compound unit, and the functor's parameter is further used in the **link** clause, since unit linking is by name. The presence of the annotations serves only to allow our translation to be a local translation. Since structure and functor definitions are first order, the translation could instead record the types of the structures and functors as it encounters their definitions. Then it could look up the types, instead of relying on the annotations.

Theorem 1 states that if a structure and functor program has a type, then the result of the translation $\Longrightarrow$ has a type that is related to the original type by the translation. We assume that source programs do not contain identifiers ending in "_unit", and we adopt the convention that for $\mathcal{X}$ in $\{\Gamma, B, D, F, \ldots\}$, if $\mathcal{X}$ and $\mathcal{X}'$ appear in the following, then $\mathcal{X}' \Longrightarrow \mathcal{X}$.

Lemma 1 (Context translation)**.** $\Gamma_1' \vdash \Gamma_2'$ implies $\Gamma_1 \vdash \Gamma_2$, and $\Gamma_1' \vdash Ty : K$ implies $\Gamma_1 \vdash Ty : K$.

Lemma 2 (Translation subtyping)**.** $\Gamma_1' \vdash Ty_1' \equiv Ty_2'$ implies $\Gamma_1 \vdash Ty_1 <: Ty_2$ and $\Gamma_1' \vdash \Gamma_2' <: \Gamma_3'$ implies $\Gamma \vdash \Gamma_2 <: \Gamma_3$.

Lemma 3 (Translation lookup)**.** If $\Gamma_1' \vdash Pm \mapsto m^i\textbf{:}\Gamma_2'$, and $\mathcal{M} = \text{STRENGTHEN}(m^i, \Gamma_2)$ **provide** $\text{DOM}_\rho(\Gamma_2)$ then $\Gamma_1 \vdash P_m\_\text{unit} \mapsto m\_\text{unit}^i\textbf{:unitT } (\text{export}^i\textbf{:}\mathcal{M})$ **(**$\epsilon \to$ export**)**

Theorem 1. If Γ_1' is well-formed ($\epsilon \vdash \Gamma_1'$) then

1. $\Gamma_1' \vdash D'_{seq} : \Gamma_2'$ implies $\Gamma_1 ; \epsilon \vdash D_{seq} : \Gamma_2$
2. $\Gamma_1' \vdash E' : Ty$ implies $\Gamma_1 \vdash E : Ty$

Proof sketch. The proof proceeds by induction on the structure of D'_{seq} and E'.

case $x^i\textbf{:}Ty\textbf{=}E'$ D'_{seq}: By lemmas 1 and 2.
case $t^i\textbf{:}K\textbf{=}Ty$ D'_{seq}: By lemma 1.
case structure m^i :> Γ' **=** D'_{seq1} D'_{seq2}: The **invoke** definition types by lemmas 1 and 2. The other unit relies on the correctness of STRENGTHEN and ΓTODS.
case structure m^i :> Γ_1' **=** $P_1\textbf{:}F'(P_2\textbf{:}\Gamma_2')$ D_{seq}: The **compound** type checks by lemmas 1, 2, and 3.
case functor $m^i\textbf{:}F'\textbf{=}D_{seq1}$ D_{seq2}: By lemmas 1 and 2. □

MkHeap: . . . = **unit import** [*tM*: $\star$,*compareM*:*t* $\to$ *t* $\to$ *order*]
export [*h*: $\star$,*item*: $\star$ =*tM*,*insert*:*item* $\to$ *h* $\to$ *h*]
h: $\star$ =*G*(*tM*)
item: $\star$ =*tM*
insert: . . . =λ . . . *compareM* . . .
Boot: . . . = **unit import** [*hB*: $\star$,*itemB*: $\star$,*insertB*:*itemB* $\to$ *hB* $\to$ *hB*]
export [*t*: $\star$ =*F*(*hB*),*compareB*:*t* $\to$ *t* $\to$ *order*]
t: $\star$ =*F*(*hB*)
compare . . . =λ . . . *insertB* . . .
Heap: . . . =
compound import [] **export** [*h*: $\star$,*insert*:*F*(*h*) $\to$ *h* $\to$ *h*]
link *MkHeap*: . . . *Boot*: . . .
where (*hB* $\leftarrow$ *h*) (*tM* $\leftarrow$ *t*) (*itemB* $\leftarrow$ *item*)
(*compareM* $\leftarrow$ *compare*) (*insertB* $\leftarrow$ *insert*)

Figure 14. Bootstrapped Heap

4.3 Cyclic Linking Dependencies

Although ML's functors do not typically support cyclic linking dependencies (recursive linking), there are several proposals for and implementations of recursive functor extensions [7, 26, 10]. Units avoid two significant problems faced by these extensions.

Avoiding Double Vision In his thesis, Dreyer describes a potentially serious problem for recursive functors that he dubs the *double vision* problem [10]; he also notices units avoid the problem. When a functor imports a function whose type contains an abstract type that is defined in the module itself, the type system needs to match up the defined type with the import's type. The inability to do so, i.e. the double-vision problem, is illustrated by the following example (which does not typecheck).

dv_unit: . . . =
unit import [*t*: $\star$ =**int**,*x*:*t*] **export** [*u*: $\star$,*y*:*u*,*f*:**int** $\to$ *u*]
u: $\star$ =**int** *y*:*u*=1
f:**int** $\to$ *u*=λ *z*:**int.** *x*+*z*
dv_cmpd: . . . =
compound import [] **export** [*u*: $\star$,*f*:**int** $\to$ *u*]
link *dv_unit*:**import** . . . **export** . . .
where (*t* $\leftarrow$ *u*) (*x* $\leftarrow$ *y*)

The intent is that the exported type *u* should be opaque to the outside, and that the imported type *t* should be known as an **int** inside of the module. However, at the **compound**, *u* is opaque, and not known to be an **int**. Changing *t*:$\star$**=int** into *t*:$\star$, would allow the compound unit to check, but results in an ill-typed body of *u*. If we use *t*: $\star$ =*u* as the import instead of *t*: $\star$ =**int**, the unit and compound unit are both well-typed. An import referring to an export this way is not supported by functors.

Bootstrapped Heap Bootstrapped heaps [25] are difficult to support with generative recursive functors. The core of bootstrapped heaps as presented by Dreyer [9]—a simplified account of Russo's example [26]—translates easily into units. The key property of bootstrapped heaps is that the type of elements contained in the heap is defined in terms of the type of heaps themselves. We can express this dependency directly by having a *Boot* unit export the type of elements *t* and import the type of a heap *hB* (figure 14). (Suppose that *G(t)* and *F(t)* are abbreviations for some type expressions involving *t*.)

A simple example from Russo's work on recursive structures illustrates what can go wrong with generative functors like *Boot* and *MkHeap*. (The syntax has been slightly modified from his paper to eliminate the nested structure). Suppose the *NatFun* functor consumes a *Bool* structure and produces a *Nat* structure and the *BoolFun* consumes a *Nat* structure to produce a *Bool*.

```
rec (Nat, Bool)
 structure Nat = NatFun(Bool)
 structure Bool = BoolFun(Nat)
```

This example produces a run-time error in Russo's system because the *Bool* structure is used before it is defined. The error arises even if the functor bodies consist entirely of syntactic values, because it arises from an ill-founded recursion at the structure-level. Russo solves the problem using eta-expansions to delay the functor applications. Essentially, the production of a structure from a functor requires an unwanted evaluation of the functor, which is delayed to an appropriate time with an eta-expansion. Because unit compounding does not create directly accessible bindings (e.g. in a module or structure), there is no need to delay the creation of those bindings.

5. Toward a Practical Language

Our model of modules and units forms the basis of a ML-like component-oriented programming language. The following additions to our model would make it a practical language.

Signatures Signature definitions enable re-use of import and export specifications for units and compound units. An ML-like signature facility would work with little modification, as long as a **where** clause for an import signature can refer to bindings in the export signatures, and *vice versa*.

Type Annotations The type annotations on value definitions and function parameters should be optional and supplied by type inference. The context annotation on **invoke** and the import/export annotation on each sub-part of a compound unit could be relaxed to simple lists of imported and exported identifiers, or they could be supplied as signatures. However, annotations should not be omitted entirely since they specify which names are introduced by the **invoke**, and which names are available as linkages in a **compound**.

Linking Specifying a linkage for each import in a compound is tedious in all but the smallest systems. Linkages should be specified between signatures, which would denote linkages of all of their components.

Renaming Our model uses stamps to differentiate between internal and external names. A practical system should provide mechanisms for explicitly managing the mapping between internal and external names on unit imports and exports.

Sealing The translation from structures to modules gives a suitable semantic account of sealing for modules. A practical language with support for structure-like sealing on modules could either use the translation, or specialize semantically equivalent functionality.

Compilation Management Compilation management in this setting entails searching the pool of top-level modules to find and compile each of a module's dependencies before compiling the module itself. To make the search unambiguous, each module in the pool must have a unique name, although in a practical setting uniqueness might instead be required at the level of file paths or URIs. This is most similar to Caml's [21] compilation management, which makes all definitions (excepting in the interactive environment) appear in a top-level structure whose name corresponds with the name of the file containing it. (Top-level structures can be sealed with signatures and used as functor arguments, just as other structures.) Haskell's and PLT Scheme's compilation management are also based on top-level modules; however, dependencies are stated explicitly by importing entire modules, instead of implicitly as the heads of paths.

Top-level Components Often, when writing highly componetized programs, top-level modules contain only the definition of a single unit. Direct support for this idiom with a top-level component construct that combines unit and module features could be implemented with a combination of our modules and units.

$\sigma : B \cup D \to ident$
the identifier defined by the binding
$\text{DOM} : \Gamma \cup DS \to ident^*$ and $\text{DOM}_\rho : \Gamma \cup DS \to name^*$
the set of identifiers or names bound by the context or definitions
$\text{DISTINCT} \subseteq \Gamma$ and $\text{DISTINCT}_\rho \subseteq \Gamma$
ensures the context does not bind an identifier or name more than once
$\Gamma \vdash P \mapsto B$
path lookup (see section 6.1)
$\Gamma \vdash ident \triangleright B; \Gamma$ and $\Gamma \vdash name \triangleright B; \Gamma$
identifier and name lookup. The returned context is the prefix of the input context where the binding was found.
$\text{PERM} \subseteq \Gamma \times \Gamma$
ensures that the contexts are permutations of each other
$\text{INTERLEAVE} \subseteq \Gamma \times \Gamma \times \Gamma$
ensures that the third argument is an interleaving of the first two
$\cdot\{ident \leftarrow E \mid \mathcal{P}(ident, E)\}$
capture-avoiding substitution of the expr./ident. pairs satisfying $\mathcal{P}$
$\text{LIMPORTS} : L^* \to ident^*$
gets the left sides of linkages

Figure 15. Helper functions and relations

6. Type System

Figure 15 describes the helper functions and relations used by the type system, and figure 16 presents the relations that define well-formed contexts, bindings, and types (the rules for sum and product types are omitted). These relations ensure that no identifier shadows another, that each type path refers to a defined type, and that module bindings and unit types are well-formed. The first restriction is for the type system's convenience; any otherwise well-formed type can meet these restrictions with α-renaming of the stamps on identifiers. The rule for module bindings (BMOD) ensures that each provided name is defined in the module, and that no name is multiply defined in a module's body. The rule for units (KUNIT) establishes that the lists of imported and exported names exactly partition the unit's context into imported and exported bindings. The import context is not separated from the export context because an import can refer to an exported type and *vice versa*.

The type rules for definitions produce bindings, and the type rules for definition sequences produce contexts (figure 17). The definition sequence rules build the context and ensure that no identifier shadows another. They have an additional context argument that the DS2 rule uses as a source for bindings. This additional context is ϵ except when type checking the body of a unit, as discussed with units below. The DVAL and DTYPE rules produce the specified type, and ensure that the resulting binding is well-formed. The DINV rule checks that its sub-expression has a unit type with no imports. The expression's type must be a subtype of the specified context, suitably placed into a unit type. The DMOD1 and DMOD2 rules ensure that the provided names are defined, and that no names are multiply defined. Additionally, DMOD2 ensures that the module's body is consistent with the module expression's declared context.

The type system for expressions (figure 18) is typical of typed λ-calculi, but with the addition of the EUNIT and ECMPD rules. Standard rules for sums and products are omitted from figure 18. The IE rule builds a unit's type from its declared imports and exports, and it uses the KUNIT rule to ensure that the resulting type is well-formed; in particular, the same name cannot be both imported and exported. Interleaving allows imports to depend upon types declared in the export portion. The EUNIT rule checks the body of the unit, using the unit's declared imports as the second context of the declaration rule. Thus, the body is checked as though the bindings of the unit's import were interspersed in the body in some order. The DROPBS function lets the EUNIT rule ignore unexported bindings (perhaps with duplicate names) when checking that the unit's body satisfies the unit's exports.

$\boxed{\Gamma \vdash \Gamma}$

$$\frac{}{\Gamma \vdash \epsilon}\ \Gamma\epsilon \qquad \frac{\Gamma_1 \vdash B \qquad \Gamma_1,B \vdash \Gamma_2 \qquad \sigma(B) \notin \text{DOM}(\Gamma_1)}{\Gamma_1 \vdash B,\Gamma_2}\ \Gamma\text{S}$$

$\boxed{\Gamma \vdash B}$

$$\frac{\Gamma \vdash Ty : \star}{\Gamma \vdash x^i\text{:}Ty}\ \text{BVAL} \qquad \frac{}{\Gamma \vdash t^i\text{:}K}\ \text{BTYPE1} \qquad \frac{\Gamma \vdash Ty : K}{\Gamma \vdash t^i\text{:}K\text{=}Ty}\ \text{BTYPE2}$$

$$\frac{\Gamma_1 \vdash \Gamma_2 \qquad names \subseteq \text{DOM}_\rho(\Gamma_2) \qquad \text{DISTINCT}_\rho(\Gamma_2)}{\Gamma_1 \vdash m^i\text{:}\Gamma_2\ \textbf{provide}\ names}\ \text{BMOD}$$

$\boxed{\Gamma \vdash Ty : K}$

$$\frac{}{\Gamma \vdash \textbf{int} : \star}\ \text{KINT} \qquad \frac{\Gamma \vdash Ty_1 : \star \qquad \Gamma \vdash Ty_2 : \star}{\Gamma \vdash Ty_1 \to Ty_2 : \star}\ \text{KFUN}$$

$$\frac{(\Gamma \vdash P_t \mapsto t^i\text{:}K) \vee (\Gamma \vdash P_t \mapsto t^i\text{:}K\text{=}Ty)}{\Gamma \vdash P_t : K}\ \text{KPATH}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Gamma_{ie} \qquad names_1 \cup names_2 = \text{DOM}_\rho(\Gamma_{ie}) \\ names_1 \cap names_2 = \emptyset \qquad \text{DISTINCT}_\rho(\Gamma_{ie})\end{array}}{\Gamma \vdash \textbf{unitT}\ \Gamma_{ie}\ (names_1 \to names_2) : \star}\ \text{KUNIT}$$

Figure 16. Well-formed types

$\boxed{\Gamma;\Gamma \vdash DS : \Gamma}$

$$\frac{}{\Gamma;\epsilon \vdash \epsilon : \epsilon}\ \text{DS}\epsilon \qquad \frac{\begin{array}{c}\Gamma_1 \vdash D : B \\ \Gamma_1,B;\Gamma_i \vdash Ds : \Gamma_2 \qquad \sigma(B) \notin \text{DOM}(\Gamma_1)\end{array}}{\Gamma_1;\Gamma_i \vdash D\ Ds : B,\Gamma_2}\ \text{DS1}$$

$$\frac{\Gamma_1,B;\Gamma_i \vdash Ds : \Gamma_2 \qquad \sigma(B) \notin \text{DOM}(\Gamma_1)}{\Gamma_1;B,\Gamma_i \vdash Ds : B,\Gamma_2}\ \text{DS2}$$

$\boxed{\Gamma \vdash D : B}$

$$\frac{\Gamma_1 \vdash E : Ty_2 \qquad \Gamma_1 \vdash Ty_2 <: Ty_1 \qquad \Gamma_1 \vdash Ty_1 : \star}{\Gamma_1 \vdash x^i\text{:}Ty_1\text{=}E : x^i\text{:}Ty_1}\ \text{DVAL}$$

$$\frac{\Gamma_1 \vdash Ty : K}{\Gamma_1 \vdash t^i\text{:}K\text{=}Ty : t^i\text{:}K\text{=}Ty}\ \text{DTYPE}$$

$$\frac{\begin{array}{c}Ty_1 = \textbf{unitT}\ \Gamma_2\ (\epsilon \to \text{DOM}_\rho(\Gamma_2)) \\ \Gamma_1 \vdash Ty_1 : \star \qquad \Gamma_1 \vdash E : Ty_2 \qquad \Gamma_1 \vdash Ty_2 <: Ty_1\end{array}}{\Gamma_1 \vdash \textbf{invoke}\ E\ \textbf{as}\ m^i\text{:}\Gamma_2 : m^i\text{:}\Gamma_2\ \textbf{provide}\ \text{DOM}_\rho(\Gamma_2)}\ \text{DINV}$$

$$\frac{\Gamma_1;\epsilon \vdash Ds : \Gamma_2 \qquad names \subseteq \text{DOM}_\rho(\Gamma_2) \qquad \text{DISTINCT}_\rho(\Gamma_2)}{\Gamma_1 \vdash \textbf{module}\ m^i\ \textbf{provide}\ names = Ds : m^i\text{:}\Gamma_2\ \textbf{provide}\ names}\ \text{DMOD1}$$

$$\frac{\begin{array}{c}\Gamma_1;\epsilon \vdash Ds : \Gamma_2 \qquad \Gamma_1 \vdash \Gamma_3 \qquad names \subseteq \text{DOM}_\rho(\Gamma_3) \\ \text{DISTINCT}_\rho(\Gamma_2) \qquad \text{DISTINCT}_\rho(\Gamma_3) \qquad \Gamma_1 \vdash \Gamma_2 <: \Gamma_3\end{array}}{\begin{array}{c}\Gamma_1 \vdash \textbf{module}\ m^i\text{:}\Gamma_3\ \textbf{provide}\ names = \\ Ds : m^i\text{:}\Gamma_3\ \textbf{provide}\ names\end{array}}\ \text{DMOD2}$$

Figure 17. Type system (definitions)

The following example demonstrates typechecking for units.

module *unit_example* **provide** *u_ex* **=**
 u_ex **: unitT** [*t*: $\star$,*v*:*t*,*w*:*t*] (*v* $\to$ *w t*) **=**
  unit import [*v*:*t*] **export** [*t*: $\star$,*w*:*t* $\to$ *t*]**.**
   t **:** $\star$ **= int**
   w **:** *t* $\to$ *t* **=** λ *x*:*t***.** *x* **+** *v*

The type of import *v* refers to the exported type *t*. When typechecking the body of the unit, the binding for *v* must be inserted after checking the definition of *t*, but before the definition of *w*. In general, each imported binding should be inserted at the earliest point where it is well-defined to do so.[2]

The ECMPD rule also uses IE to build the result type, and uses DROPBS to ignore unneeded exports from the sub-units when checking the exports from the compound unit. However, the process for creating the context for the compound unit's body, Γ_4, is more involved than for non-compound units. The US check ensures that the type of each sub-unit expression is compatible with the corresponding import and export contexts. The DISTINCT check ensures that there is no ambiguity in what each linkage refers to, and the check involving LIMPORTS ensures that each sub-unit import is linked to exactly once.

The substitution applies the linkages to the context of the sub-units' exports, redirecting type and module references from the sub-unit's import to the exported definition that satisfies the import. The L rule ensures that the type of a linkage's export is a subtype of its import. Since the type of an import can refer to types and modules defined in other imports (of the same sub-unit), the L rule performs linkage substitution on the import before comparing it to the corresponding export.

The following example (which includes the identifier stamps), demonstrates typechecking for compound units.

module *compound_example* **provide =**
 u^1 **: unitT** [t^2: $\star$, w^1**:int** $\to$ t^2] (*t* $\to$ *w*) **=**
  unit import [t^2: $\star$] **export** [w^1**:int** $\to$ t^2]**.** ...
 u^2 **: unitT** [u^4: $\star$ **=int**] ($\to$ *u*) **=**
  unit import [] **export** [u^2: $\star$ **=int**]**.** ...
 u^3 **: unitT** [w^1**:int** $\to$ **int**] ($\to$ *w*) **=**
  compound import [] **export** [w^1**:int** $\to$ **int**]**.**
   link u^2 **: import** [] **export** [u^4: $\star$ **=int**]
        u^1 **: import** [t^2: $\star$] **export** [w^1**:int** $\to$ t^2]
   where t^2 $\leftarrow$ u^4

The intermediate context Γ_3 is **[**u^4**:** $\star$**=int,**w^1**:int** $\to$ t^2**]**, so that the context after substitution is Γ'_3 = **[**u^4**:** $\star$ **=int,**w^1**:int** $\to$ u^4**]**.

The permutation of Γ_i**,**Γ'_3 allows the sub-units to have mutual type dependencies while stopping short of supporting a true recursive type environment, and equi-recursive types. In the following interpreter-inspired example, suppose that the types of the two *run* functions refer to the *dec* and exp types, and that these two types depend on each other. Then in Γ_4, the bindings for exp^2 and dec^1 must both precede the bindings for *runD* and *runE*.

exp_u: ... **=unit import** [*dec*:$\star$,*runD*: ...] **export** [*exp*:$\star$,*runE*: ...]**.** ...
dec_u: ... **=unit import** [*exp*:$\star$,*runE*: ...] **export** [*dec*:$\star$,*runD*: ...]**.** ...
run_u: ... **=**
 compound import [] **export** [dec^2: $\star$,exp^1: $\star$,$runE^1$: ... ,$runD^2$: ...]
  link *exp_u*:**import** [dec^1: $\star$,$runD^1$: ...] **export** [exp^1: $\star$,$runE^1$: ...]
       dec_u:**import** [exp^2: $\star$,$runE^2$: ...] **export** [dec^2: $\star$,$runD^2$: ...]
  where (dec^1 $\leftarrow$ dec^2) ($runD^1$ $\leftarrow$ $runD^2$)
        (exp^2 $\leftarrow$ exp^1) ($runE^2$ $\leftarrow$ $runE^1$)

If the *exp* and *dec* types were exported translucently, exposing the

[2] Modules provide a convenient mechanism for grouping imports and exports. However, they have limited flexibility for recursive dependencies, because the entire module binding must appear atomically, which limits interleaving. In the example, if *t* and *w* were grouped in a module, *v* could not be interspersed. This restriction could be relaxed by letting the unit rule split the module body's context throughout its enclosing context, or by using a signature mechanism as mentioned in section 5 (instead of modules) to group imports and exports.

$\Gamma \vdash \Gamma$ DROPBS Γ
drop bindings from the middle context

$$\boxed{\Gamma \vdash E : Ty}$$

$$\frac{\Gamma \vdash P_X \mapsto x^i{:}Ty}{\Gamma \vdash P_X : Ty}\ \textsc{Epath} \qquad \frac{\Gamma{,}x^i{:}Ty_1 \vdash E : Ty_2 \quad \Gamma \vdash Ty_1 : \star \quad x^i \notin \textsc{dom}(\Gamma)}{\Gamma \vdash \lambda x^i{:}Ty_1.E : Ty_1 \to Ty_2}\ \textsc{Efun}$$

$$\frac{\Gamma \vdash_{\mathrm{P}} E_1 : Ty_2 \to Ty_3 \quad \Gamma \vdash E_2 : Ty_4 \quad \Gamma \vdash Ty_4 <: Ty_2}{\Gamma \vdash E_1\ E_2 : Ty_3}\ \textsc{Eapp}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \textbf{import}\ \Gamma_{\mathrm{i}}\ \textbf{export}\ \Gamma_{\mathrm{e}} : Ty \\ \Gamma_1; \Gamma_{\mathrm{i}} \vdash Ds : \Gamma_2 \quad \Gamma_1 \vdash \Gamma_2\ \textsc{dropbs}\ \Gamma_3 \\ \textsc{dom}(\Gamma_3) = \textsc{dom}(\Gamma_{\mathrm{i}}) \cup \textsc{dom}(\Gamma_{\mathrm{e}}) \quad \textsc{distinct}_\rho(\Gamma_3) \\ \Gamma_1 \vdash \textbf{unitT}\ \Gamma_3\ (\textsc{dom}_\rho(\Gamma_{\mathrm{i}}) \to \textsc{dom}_\rho(\Gamma_{\mathrm{e}})) <: Ty\end{array}}{\Gamma_1 \vdash \textbf{unit import}\ \Gamma_{\mathrm{i}}\ \textbf{export}\ \Gamma_{\mathrm{e}}.\ Ds : Ty}\ \textsc{Eunit}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \textbf{import}\ \Gamma_{\mathrm{i}}\ \textbf{export}\ \Gamma_{\mathrm{e}} : Ty \quad \Gamma_1 \vdash Us : \Gamma_2; \Gamma_3 \\ \textsc{distinct}(\Gamma_{\mathrm{i}}{,}\Gamma_2{,}\Gamma_3) \quad \textsc{perm Limports}(Ls)\ \textsc{dom}(\Gamma_2) \\ \Gamma_3' = \Gamma_3\{id_1 \leftarrow id_2 \mid (id_1 \leftarrow id_2) \in Ls\} \\ \textsc{perm}\ (\Gamma_{\mathrm{i}}{,}\Gamma_3')\ \Gamma_4 \quad \Gamma_1 \vdash \Gamma_4 \\ \forall L.\, L \in Ls \Rightarrow \Gamma_1{,}\Gamma_4; \Gamma_2; Ls \vdash L \quad \Gamma_1 \vdash \Gamma_4\ \textsc{dropbs}\ \Gamma_5 \\ \textsc{dom}(\Gamma_5) = \textsc{dom}(\Gamma_{\mathrm{i}}) \cup \textsc{dom}(\Gamma_{\mathrm{e}}) \quad \textsc{distinct}_\rho(\Gamma_5) \\ \Gamma_1 \vdash \textbf{unitT}\ \Gamma_5\ (\textsc{dom}_\rho(\Gamma_{\mathrm{i}}) \to \textsc{dom}_\rho(\Gamma_{\mathrm{e}})) <: Ty\end{array}}{\Gamma_1 \vdash \textbf{compound import}\ \Gamma_{\mathrm{i}}\ \textbf{export}\ \Gamma_{\mathrm{e}}\ \textbf{link}\ Us\ \textbf{where}\ Ls : Ty}\ \textsc{Ecmpd}$$

$$\boxed{\Gamma \vdash \textbf{import}\ \Gamma\ \textbf{export}\ \Gamma : Ty}$$

$$\frac{\textsc{interleave}\ \Gamma_{\mathrm{i}}\ \Gamma_{\mathrm{e}}\ \Gamma_{\mathrm{ie}} \quad Ty = \textbf{unitT}\ \Gamma_{\mathrm{ie}}\ (\textsc{dom}_\rho(\Gamma_{\mathrm{i}}) \to \textsc{dom}_\rho(\Gamma_{\mathrm{e}})) \quad \Gamma \vdash Ty : \star}{\Gamma \vdash \textbf{import}\ \Gamma_{\mathrm{i}}\ \textbf{export}\ \Gamma_{\mathrm{e}} : Ty}\ \textsc{Ie}$$

$$\boxed{\Gamma \vdash U^* : \Gamma; \Gamma}$$

$$\frac{}{\Gamma \vdash \epsilon : \epsilon; \epsilon}\ \textsc{Us}\epsilon \qquad \frac{\Gamma_1 \vdash \textbf{import}\ \Gamma_{\mathrm{i}}\ \textbf{export}\ \Gamma_{\mathrm{e}} : Ty_1 \quad \Gamma_1 \vdash E : Ty_2 \quad \Gamma_1 \vdash Ty_2 <: Ty_1 \quad \Gamma_1 \vdash Us : \Gamma_2; \Gamma_3}{\Gamma_1 \vdash (E{:}\textbf{import}\ \Gamma_{\mathrm{i}}\ \textbf{export}\ \Gamma_{\mathrm{e}})\ Us : \Gamma_{\mathrm{i}}{,}\Gamma_2; \Gamma_{\mathrm{e}}{,}\Gamma_3}\ \textsc{Us}$$

$$\boxed{\Gamma; \Gamma; L^* \vdash L}$$

$$\frac{\Gamma_{\mathrm{i}} \vdash id_1 \triangleright B_1; \Gamma_{\mathrm{i}}' \quad \Gamma \vdash id_2 \triangleright B_2; \Gamma' \quad B_1' = B_1\{id_1 \leftarrow id_2 \mid (id_1 \leftarrow id_2) \in Ls\} \quad \Gamma \vdash B_2 <: B_1'}{\Gamma; \Gamma_{\mathrm{i}}; Ls \vdash id_1 \leftarrow id_2}\ \textsc{L}$$

Figure 18. Type system (expressions)

mutual reference, no ordering would allow *run* to typecheck, since *exp* would need to precede *dec* and simultaneously, *exp* would need to precede *dec*.

6.1 Path Lookup

When looking up a path in a context (figure 19), any types in the returned binding are lifted out of their scope. Paths that are contained within the type itself, and that reference bindings in their enclosing module, would then become free references, or get captured by an intervening binding. The substitution prevents this by changing these identifiers into paths to their original binding. The substitution must use hatted names in the path, so the reference can succeed even if the name is not provided. The definitions of lookup for P_t and P_X paths are similar.

$$\boxed{\widehat{P_m} = P_m}$$

$$\begin{array}{rcl}\widehat{m^i} & = & m^i \\ \widehat{P_m.m} & = & \widehat{P_m}.\hat{m} \\ \widehat{P_m.\hat{m}} & = & \widehat{P_m}.\hat{m}\end{array}$$

$$\boxed{\mathcal{M} \vdash P_m \mapsto B}$$

$$\frac{\Gamma \vdash m^i \triangleright B; \Gamma'}{\Gamma \vdash m^i \mapsto B}\ \textsc{Pid}$$

$$\frac{\Gamma_1 \vdash P \mapsto m_2^i{:}\Gamma_2\ \textbf{provide}\ names_2 \quad m \in names_2 \quad \Gamma_2 \vdash m \triangleright B; \Gamma_2'}{\Gamma_1 \vdash P.m \mapsto B\{m_3^i \leftarrow \widehat{P.m} \mid m_3^i \in \textsc{dom}(\Gamma_2')\}}\ \textsc{Pname}$$

$$\frac{\Gamma_1 \vdash P \mapsto m_2^i{:}\Gamma_2\ \textbf{provide}\ names_2 \quad \Gamma_2 \vdash m \triangleright B; \Gamma_2'}{\Gamma_1 \vdash P.\hat{m} \mapsto B\{m_3^i \leftarrow \widehat{P.m} \mid m_3^i \in \textsc{dom}(\Gamma_2')\}}\ \textsc{Phname}$$

Figure 19. Path lookup

For example, with the following module definition the path $m^1.n.v$ has type $m^1.\hat{t} \to m^1.\hat{n}.\hat{u}$, which is equal to **int** → **int**.

module m^1 **provide** n **=**
t^2**:** $\star$ **= int**
module n^3 **provide** v **=**
u^4**:** $\star$ **= int**
v^5**:**$t^2 \to u^4$ **=** ...

Without the hatted identifiers, the result type would be $m^1.t \to m^1.n.u$ which refers to names that are not provided by the module, and are hence inaccessible through the PNAME rule.

6.2 Subtyping

Subtyping is based on a simple structural subtyping relation (figure 20; not shown are the typical subtyping rules for sum, product and function types). A context Γ_1 is a sub-context of Γ_2 if each identifier bound in Γ_2 appears in Γ_1, bound to a sub-binding. Binding subtypes make a transparent type binding a subtype of an opaque type binding; however two transparent type bindings must have equal types. This is because references to the transparent type could occur in both co– and contra-variant positions. Sub-typing for unit types is similar to context sub-typing, but based on the defined names instead of identifiers. Unit sub-typing is contra-variant in the imports and co-variant in the exports.

Type paths (P_t) are structural subtypes only if they are equal, but the general subtyping relations ($\Gamma \vdash Ty <: Ty$, $\Gamma \vdash B <: B$, $\Gamma \vdash \Gamma <: \Gamma$) first allow type paths to be replaced with their definitions, and then they check structural subtyping.

7. Operational Semantics and Type Soundness

We specify the operational semantics of units as a small-step reduction relation with call-by-value evaluation. Figure 21 presents the values and evaluation contexts for programs (*PV*, *PC*), modules (*MV*, *MC*), definition sequences (*SV*, *SC*), definitions (*DC*, *DV*), expressions (*V*, *EC*), and the subexpressions of **compound** (*UC*, *UV*). The evaluation strategy accumulates evaluated modules and definitions as it progresses through the program.

Figure 22 describes helper functions and relations used in the operational semantics and soundness proof, and figure 23 presents the single-step reduction relation $\rightsquigarrow$. The rules for projections, function application, and case expressions are straightforward. Unit invocation places the unit's body into a module, and unit com-

$$\boxed{Ty <: Ty}$$

$$\frac{\widehat{P_{t1}} = \widehat{P_{t2}}}{P_{t1} <: P_{t2}}$$

$$\frac{\begin{array}{c} names_1 \subseteq names_3 \qquad names_4 \subseteq names_2 \\ names_1 \subseteq \text{DOM}_\rho(\Gamma_2) \qquad \text{DOM}_\rho(\Gamma_1) \subseteq names_4 \\ \forall n \in names_1.\ (\Gamma_1 \vdash n \triangleright B_1; \Gamma_1') \wedge (\Gamma_2 \vdash n \triangleright B_2; \Gamma_2') \Rightarrow (B_2 <: B_1) \\ \forall n \in names_4.\ (\Gamma_1 \vdash n \triangleright B_1; \Gamma_1') \wedge (\Gamma_2 \vdash n \triangleright B_2; \Gamma_2') \Rightarrow (B_1 <: B_2) \end{array}}{\textbf{unitT}\ \Gamma_1\ (names_1 \to names_2) <: \textbf{unitT}\ \Gamma_2\ (names_3 \to names_4)}$$

$$\boxed{\Gamma <: \Gamma}$$

$$\frac{\begin{array}{c} \text{DOM}(\Gamma_2) = ids \qquad ids \subseteq \text{DOM}(\Gamma_1) \\ \forall id \in ids.\ (\Gamma_1 \vdash id \triangleright B_1; \Gamma_1') \wedge (\Gamma_2 \vdash id \triangleright B_2; \Gamma_2') \Rightarrow (B_1 <: B_2) \end{array}}{\Gamma_1 <: \Gamma_2}$$

$$\boxed{B <: B}$$

$$\frac{}{t_1^i{:}K <: t_2^i{:}K} \qquad \frac{}{t_1^i{:}K{=}Ty <: t_2^i{:}K{=}Ty} \qquad \frac{}{t_1^i{:}K{=}Ty <: t_2^i{:}K}$$

$$\frac{Ty_1 <: Ty_2}{x_1^i{:}Ty_1 <: x_2^i{:}Ty_2} \qquad \frac{names_2 \subseteq names_1 \qquad \Gamma_1 <: \Gamma_2}{m_1^i{:}\Gamma_1\ \textbf{provide}\ names_1 <: m_2^i{:}\Gamma_2\ \textbf{provide}\ names_2}$$

Figure 20. Structural subtyping

PV	=	MV^*
MV	=	**module** m^i **provide** $name^*$ **=** SV
	|	**module** $m^i{:}\Gamma$ **provide** $name^*$ **=** SV
SV	=	ϵ | $DV\ SV$ | (**rec** SV) SV
DV	=	$x^i{:}Ty{=}V$ | $t^i{:}K{=}Ty$ | MV
V	=	$\mathbb{Z}$ | **injl** V | **injr** V | (V, V) | $\lambda x^i{:}Ty.E$
	|	**unit import** Γ **export** Γ**.** DS
UV	=	V**:import** Γ **export** Γ
PC	=	$MV^*\ MC\ M^*$
MC	=	**module** m^i **provide** $name^*$ **=** SC
	|	**module** $m^i{:}\Gamma$ **provide** $name^*$ **=** SC
SC	=	[] | $DC\ DS$ | $DV\ SC$ | (**rec** SC) DS | (**rec** SV) SC
DC	=	[] | $x^i{:}Ty{=}EC$ | **invoke** EC **as** $m^i{:}\Gamma$ | MC
EC	=	[] | **injl** EC | **injr** EC
	|	**case** EC **of** E **+** E | **case** V **of** EC **+** E | **case** V **of** V **+** EC
	|	(EC, E) | (V, EC) | $\pi_1\ EC$ | $\pi_2\ EC$ | $EC\ E$ | $V\ EC$
	|	**compound import** Γ **export** Γ **link** $UV^*\ UC\ U^*$ **where** L^*
UC	=	EC**:import** Γ **export** Γ

Figure 21. Values and evaluation contexts

pounding concatenates the bodies of the linked units into a single unit body. Because unit linking can be recursive, the resulting unit can contain recursive value, type, and module definitions.

The type system of section 6 does not support the recursive definitions (**rec**), so we extend it in figure 24 to allow the result of RCMPD to be well-typed. Any well-typed program in the system of section 6 is also well-typed in the extension, since the extension just adds new rules for the new construct. Thus, type soundness for the extended system implies soundness for the original system in the sense that a well-typed program will not become stuck. However, the final result of evaluation might not be well-typed in the original system, since it can contain **rec** statements introduced by RCMPD.[3]

[3] We chose the system of section 6 to demonstrate that units can be type checked without special support for either value or type recursion, other than what the units provide themselves. In particular, units can be type checked without encountering any of the difficulties presented by equi-recursive types. The dynamic semantics must support evaluation of definition sequences that contain **rec** constructs, but the type checking of intermediate steps in program execution is only needed for the inductive preservation and progress proofs.

FLAT : $PC \times (DS \cup D \cup E) \to DS$
flattens the evaluation context supposing that the second argument would be used to fill the hole

$DS \vdash P_X \mapsto D \cup \{\textbf{error}\}$
path lookup in a definition sequence. Returns **error** if the path goes through an **invoke** definition.

CTXT : $DS \to \Gamma$ and CTXT : $PC \times (Ds \cup D \cup E) \to \Gamma$
get the definitions' corresponding context. The necessary information is syntactically available. The second version performs FLAT first.

$\Gamma \vdash_R \Gamma$ and $\Gamma \vdash_R DS : \Gamma$
type checking for sequences that appear as the body of a recursive definition sequence. Similar to the corresponding relations in figures 16 and 17, except these do not accumulate bindings for checking bindings/definitions later in the sequence.

Figure 22. Operational semantics helper functions and relations

$$\boxed{Prog \rightsquigarrow Prog}$$

$$\frac{names = \text{DOM}_\rho(\Gamma_2) \qquad \text{DISTINCT}_\rho(Ds)}{\begin{array}{c} PC[\textbf{invoke}\ (\textbf{unit import}\ \epsilon\ \textbf{export}\ \Gamma_1\textbf{.}\ Ds)\ \textbf{as}\ m^i{:}\Gamma_2] \rightsquigarrow \\ PC[\textbf{module}\ m^i{:}\Gamma_2\ \textbf{provide}\ names\ \textbf{=}\ Ds] \end{array}}\ \text{RINV}$$

$$\frac{\text{FLAT}(PC, P_X) \vdash P_X \mapsto x^i{:}Ty{=}E \qquad E \in V}{PC[P_X] \rightsquigarrow PC[E]}\ \text{RPATH}$$

$$\frac{\text{FLAT}(PC, P_X) \vdash P_X \mapsto x^i{:}Ty{=}E \qquad E \notin V}{PC[P_X] \rightsquigarrow \textbf{error}}\ \text{RERR1}$$

$$\frac{\text{FLAT}(PC, P_X) \vdash P_X \mapsto \textbf{error}}{PC[P_X] \rightsquigarrow \textbf{error}}\ \text{RERR2}$$

$$PC[\pi_1(V_1, V_2)] \rightsquigarrow PC[V_1]\ \text{RPJ1} \qquad PC[\pi_2(V_1, V_2)] \rightsquigarrow PC[V_2]\ \text{RPJ2}$$

$$PC[\textbf{case injl}\ V_1\ \textbf{of}\ V_2\ \textbf{+}\ V_3] \rightsquigarrow PC[V_2\ V_1]\ \text{RCASE1}$$

$$PC[\textbf{case injr}\ V_1\ \textbf{of}\ V_2\ \textbf{+}\ V_3] \rightsquigarrow PC[V_3\ V_1]\ \text{RCASE2}$$

$$PC[(\lambda x^i{:}Ty.E)\ V] \rightsquigarrow PC[E\{x^i \leftarrow V\}]\ \text{RAPP}$$

$$\frac{\begin{array}{c} UVs = (E_1\textbf{:import}\ \Gamma_{1,3}\ \textbf{export}\ \Gamma_{1,4}) \cdots \\ \forall j > 0.E_j = \textbf{unit import}\ \Gamma_{j,1}\ \textbf{export}\ \Gamma_{j,2}\textbf{.}\ Ds_j \\ \forall j > 0.(\text{DOM}(\Gamma_{j,1}) \subseteq \text{DOM}(\Gamma_{j,3}) \wedge \text{DOM}(\Gamma_{j,4}) \subseteq \text{DOM}(\Gamma_{j,2})) \\ Ds' = Ds_1 \cdots \qquad \text{DISTINCT}(\Gamma_{0,1}, \text{CTXT}(Ds')) \\ Ds'' = Ds'\{id_1 \leftarrow id_2 \mid (id_1 \leftarrow id_2) \in Ls\} \end{array}}{\begin{array}{c} PC[\textbf{compound import}\ \Gamma_{0,1}\ \textbf{export}\ \Gamma_{0,2}\ \textbf{link}\ UVs\ \textbf{where}\ Ls] \rightsquigarrow \\ PC[\textbf{unit import}\ \Gamma_{0,1}\ \textbf{export}\ \Gamma_{0,2}\textbf{.}\ \textbf{rec}\ Ds''] \end{array}}\ \text{RCPD}$$

Figure 23. One-step reduction

The path lookup function adds prefixes to the values it lifts out of modules, just like the type system's path lookup of section 6.1. If evaluation encounters a non-value expression, it signals a run-time error (RERR1 and RERR2). There are many proposals to catch such ill-founded recursion errors at compile-time without restricting definitions to syntactic values [5, 8, 15, 16, 29]. Adapting these proposals to component programming with units is future work.

Taking a closer look at the first compound unit rule, it requires that the unit values' imported and exported identifiers match up with the ones specified in the import and export annotations, which themselves correspond to the identifiers used in the linkages. Fur-

$$DS \quad = \quad (\textbf{rec}\ DS)\ DS \mid \ldots \qquad\qquad \Gamma \quad = \quad \Gamma\textbf{,rec}\ \Gamma \mid \ldots$$

$$\boxed{\Gamma \vdash \Gamma}$$

$$\frac{\begin{array}{c}\textsc{dom}(\Gamma_2) \cap \textsc{dom}(\Gamma_1) = \emptyset \\ \Gamma_1\textbf{,rec}\ \Gamma_2 \vdash_{\mathrm{R}} \Gamma_2 \qquad \Gamma_1\textbf{,rec}\ \Gamma_2 \vdash \Gamma_3 \qquad \textsc{distinct}(\Gamma_2)\end{array}}{\Gamma_1 \vdash \textbf{rec}\ \Gamma_2\textbf{,}\Gamma_3}\ \textsc{Grec}$$

$$\boxed{\Gamma; \Gamma \vdash DS : \Gamma}$$

$$\frac{\begin{array}{c}\Gamma_2 = \Gamma_i\textbf{,}\textsc{ctxt}(Ds_1) \\ \textsc{dom}(\Gamma_2) \cap \textsc{dom}(\Gamma_1) = \emptyset \qquad \textsc{distinct}(\Gamma_2) \\ \Gamma_1\textbf{,rec}\ \Gamma_2 \vdash_{\mathrm{R}} Ds_1 : \textsc{ctxt}(Ds_1) \qquad \Gamma_1\textbf{,rec}\ \Gamma_2; \Gamma_i' \vdash Ds_2 : \Gamma_3\end{array}}{\Gamma_1; \Gamma_i\textbf{,}\Gamma_i' \vdash (\textbf{rec}\ Ds_1)\ Ds_2 : \textbf{rec}\ \Gamma_2\textbf{,}\Gamma_3}\ \textsc{DsRec}$$

Figure 24. Type system extension

thermore, the definitions inside the units must all have distinct identifiers. These conditions can be met through α-renaming.

Theorem 2 (Type Soundness)**.** If $\epsilon; \epsilon \vdash Prog_1 : \Gamma$, and $Prog_1 \twoheadrightarrow Prog_2$ then either $Prog_2 = \textbf{error}$, or $\epsilon; \epsilon \vdash Prog_2 : \Gamma$ and either

- $Prog_2 \in PV$, or
- $Prog_2 \rightsquigarrow \textbf{error}$, or
- there exists a $Prog_3$ such that $Prog_2 \rightsquigarrow Prog_3$.

Lemma 4 (Value path lookup)**.** If $\Gamma = \textsc{ctxt}(PC, P_X)$, and $\Gamma \vdash P_X \mapsto x^i\textbf{:}Ty$ then either

- There exists an E such that $\textsc{flat}(PC, P_X) \vdash P_X \mapsto x^i\textbf{:}Ty\textbf{=}E$, or
- $\textsc{flat}(PC, P_X) \vdash P_X \mapsto \textbf{error}$.

Lemma 5 (Context lookup subtype)**.** If $\epsilon \vdash PC[P_X] : \Gamma_2$, and $\Gamma_1 = \textsc{ctxt}(PC, P_X)$, and $PC \vdash P_X \mapsto x^i\textbf{:}Ty_1\textbf{=}E$ then $\Gamma_1 \vdash E : Ty_2$ and $\Gamma_1 \vdash Ty_2 <: Ty_1$.

Lemma 6 (Subtype substitution)**.** If $\Gamma \vdash ident_1 \triangleright B_1; \Gamma'$, and $\Gamma \vdash ident_2 \triangleright B_2; \Gamma'$, and $\Gamma' \vdash B_2 <: B_1$ then $\Gamma' \vdash Ds\{ident_1 \leftarrow ident_2\} <: Ds$.

Definition 1. A *recursive* evaluation context is an evaluation context of the form $PC[(\textbf{rec}\ DV^*\ [\,]\ DS)\ DS]$.

Lemma 7 (Progress)**.**

1. If $\Gamma; \epsilon \vdash Ds_1 : \Gamma_1$, and $\Gamma = \textsc{ctxt}(PC, Ds_1)$, and PC is not recursive, then either $Ds_1 \in SV$, or there exists Ds_2 such that $PC[Ds_1] \rightsquigarrow PC[Ds_2]$, or $PC[Ds_1] \rightsquigarrow \textbf{error}$.
2. If $\Gamma \vdash_{\mathrm{R}} Ds_1 : \Gamma_1$, and $\Gamma = \textsc{ctxt}(PC, Ds_1)$, and PC is recursive, then either $Ds_1 \in SV$, or there exists Ds_2 such that $PC[Ds_1] \rightsquigarrow PC[Ds_2]$, or $PC[Ds_1] \rightsquigarrow \textbf{error}$.
3. If $\Gamma \vdash D_1 : B$, and $\Gamma = \textsc{ctxt}(PC, D_1)$ then either $D_1 \in DV$, or there exists D_2 such that $PC[D_1] \rightsquigarrow PC[D_2]$, or $PC[D_1] \rightsquigarrow \textbf{error}$.
4. If $\Gamma \vdash E_1 : Ty$, and $\Gamma = \textsc{ctxt}(PC, E_1)$ then either $E_1 \in V$, or there exists E_2 such that $PC[E_1] \rightsquigarrow PC[E_2]$, or $PC[E_1] \rightsquigarrow \textbf{error}$.

Proof sketch. Statements 1–4 are proved simultaneously by induction on the derivation that Ds_1, D_1, and E_1 are well-typed.
case DINV: To apply rule RINV, the invoked unit's body must contain no duplicate names. The body of the unit can be α-renamed to meet this. (Recall that the α relation can change the names of definitions that are purely internal to a unit.)
case EPATH: By lemma 4.
case ECMPD: To apply rule RCPD, the various context domains must match up, and the units' bodies must not contain duplicate identifier definitions among them. Since the ECMPD rule ensures that all the context annotations meet these conditions, the unit values can be α renamed to meet them. □

Lemma 8 (Preservation)**.**

1. $\Gamma; \epsilon \vdash Ds_1 : \Gamma_1$, and $\Gamma = \textsc{ctxt}(PC, Ds_1)$, and PC is not a recursive context, and $\epsilon; \epsilon \vdash \textsc{flat}(PC, Ds_1) : \Gamma$, and $PC[Ds_1] \rightsquigarrow PC[Ds_2]$ implies $\Gamma; \epsilon \vdash Ds_2 : \Gamma_1$.
2. $\Gamma \vdash_{\mathrm{R}} Ds_1 : \Gamma_1$, and $\Gamma = \textsc{ctxt}(PC, Ds_1)$, and PC is a recursive context, and $\epsilon; \epsilon \vdash \textsc{flat}(PC, Ds_1) : \Gamma$, and $PC[Ds_1] \rightsquigarrow PC[Ds_2]$ implies $\Gamma \vdash_{\mathrm{R}} Ds_2 : \Gamma_1$.
3. $\Gamma \vdash D_1 : B_1$, and $\Gamma = \textsc{ctxt}(PC, D_1)$, and $\epsilon; \epsilon \vdash \textsc{flat}(PC, D_1) : \Gamma$, and $PC[D_1] \rightsquigarrow PC[D_2]$ implies $\Gamma \vdash D_2 : B_1$.
4. $\Gamma \vdash E_1 : Ty_1$, and $\Gamma = \textsc{ctxt}(PC, E_1)$, and $\epsilon; \epsilon \vdash \textsc{flat}(PC, E_1) : \Gamma$, and $PC[E_1] \rightsquigarrow PC[E_2]$ implies $\Gamma \vdash E_2 : Ty_2$, and $\Gamma \vdash Ty_2 <: Ty_1$.

Proof sketch. Statements 1–4 are proved simultaneously by induction on the derivation that Ds_1, D_1, and E_1 are well-typed.
case EPATH: By lemma 4, the fact that path lookup in a well-formed context is deterministic, and by lemma 5.
case ECMPD: After RCPD converts a compound unit into non-compound unit, the new unit's body has a type because each of the linkage substitutions replaces an identifier with one bound to a subtype, as checked by the L rule. Thus lemma 6 applies. The ordering of definitions in the body does not matter because they are in a single **rec** definition. The context of the body's definition is a subtype of the export annotations' associated contexts, and the ECMPD rule checks that that context satisfies the resulting unit's export clause. □

8. Related Work

Dreyer et al. [11] provide a definitive model of ML module systems, but it does not include support for cyclic dependencies. Type-theoretic work on cyclic dependencies for ML has yielded the notion of recursive dependent signatures [7] and an account of recursive type generativity that resembles the backpatching semantics of Scheme recursion [9]. Russo adapts the former to a practical extension of SML with recursive structures [26]; since functor linking remains tied to functor invocation in this system, mutually-recursive functions across functor boundaries work only with an eta expansion, as discussed in section 4.3.

Dreyer's dissertation [10] provides a critique of units in comparison to ML modules and approaches to recursive functors. His comparison notes the problems with units that we address here with the first-order module construct and translucent types, and he also notes that units naturally avoid the "double-vision" problem. His dissertation also contains a recursive module system that solves these problems, but does not support separate compilation.

Harper and Lillibridge [14] introduced translucent sums at the same time as Leroy's manifest types [20]. These systems introduced the kind of translucency added to units in this paper, but neither system supports recursive linking. Harper's system supports first-class functors that obey a phase distinction allowing type checking to fully precede evaluation. Units are first-class values in the same way as these functors; however, we require that each unit's imports and exports appear in the source program (possibly using signatures), which mitigates the decidability problems encountered in first-class translucent sums.

Mixin modules [12, 17, 22, 32] support the separation of mutually recursive program values across modules. Like units, mixin modules separate the composition operation from the invocation operation. Ancona and Zucca define a core calculus for module systems with higher-order and recursive features [3], and their line of work has produced a module system for Java [2]. More recent work on polymorphic bytecode [1] addresses both direct and indirect references, much like structures and functors. This work is similar in spirit to ours, but with no connection to SML-style concerns

such as translucency, sharing by specification, etc. Scala [24] supports component programming with a class-, object-, and mixin-based approach that has a much different flavor than the ML/unit approach.

Many programming languages have a module system that manages a global namespace and supports direct reference among the modules, including Haskell [18], Ocaml (top-level structures) [21], and Java (called packages). Often read-eval-print-loop-based languages have module systems that support direct reference, but not a global namespace. These languages include Bigloo Scheme [27], Scheme48 [19], Chez Scheme[31], and structures in SML [23] (various extensions to SML add a global namespace of compilation units [4, 6, 28]). Our work recognizes the importance of supporting such a module system.

9. Conclusion

Our model of units and modules supports translucent and opaque type imports and exports, independent unit compilation, and recursive unit linking that avoids ill-founded recursive module definitions without requiring manual intervention (although recursive value definitions could be ill-founded). We are unaware of any other module system that satisfies these criteria.

Our model accounts for internal linking and external linking as orthogonal concepts. Although external linking offers the greatest flexibility for code re-use, programming without any internal linking is infeasible. For example, the *fully functorized* style of ML programming, which requires external linkages that are performed by functor application, enforces a high overhead on the programmer in cases where parameterization is unnecessary. Furthermore, the system that manages the externally linked entities must support direct references and internal linking, so that the programmer can refer to the particular components that he wishes to externally link.

The extent to which a practical language emphasizes internal or external linking is an important design decision. In PLT Scheme, top-level modules support compilation management and direct references, and unit values can appear freely in the program. This allows the programmer freedom in choosing the amount of either style of linking. For cases when a heavily component-oriented style is desired, we have implemented a top-level construct that is a combination of both unit and module. We compile this construct into a module that contains a unit, separating out its aspects into the (untyped version) of our model of modules and units.

References

[1] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: compositional compilation for Java-like languages. In *Proc. Principles of Programming Languages*, 2005.

[2] D. Ancona and E. Zucca. True modules for Java-like languages. In *Proc. European Conference on Object-Oriented Programming*, 2001.

[3] D. Ancona and E. Zucca. A calculus of module systems. *J. Funct. Program.*, 12(2):91–132, 2002.

[4] M. Blume and A. W. Appel. Hierarchical modularity. *ACM Trans. Program. Lang. Syst.*, 21(4):813–847, 1999.

[5] G. Boudol. The recursive record semantics of objects revisited. *J. Funct. Program.*, 14(3):263–315, 2004.

[6] H. Cejtin, M. Fluet, S. Jagannathan, and S. Weeks. Formal specification of the ML basis system. `http://mlton.org/pages/MLBasis/attachments/mlb-formal.pdf`, January 2005.

[7] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. Programming Language Design and Implementation*, 1999.

[8] D. Dreyer. A type system for well-founded recursion. In *Proc. Principles of Programming Languages*, 2004.

[9] D. Dreyer. Recursive type generativity. In *Proc. International Conference on Functional Programming*, 2005.

[10] D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, 2005.

[11] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Proc. Principles of Programming Languages*, 2003.

[12] D. Duggan and C. Sourelis. Mixin modules. In *Proc. International Conference on Functional Programming*, 1996.

[13] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. Programming Language Design and Implementation*, 1998.

[14] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. Principles of Programming Languages*, 1994.

[15] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. *ACM Trans. Program. Lang. Syst.*, 27(5):857–881, 2005.

[16] T. Hirschowitz, X. Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *Proc. Principles and Practice of Declaritive Programming*, 2003.

[17] T. Hirschowitz, X. Leroy, and J. B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *Proc. The European Symposium on Programming*, 2004.

[18] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[19] R. Kelsey, J. Rees, and M. Sperber. *Scheme48 Reference Manual*, 1.3 edition, 2005. `http://s48.org/1.3/s48manual.pdf`.

[20] X. Leroy. Manifest types, modules, and separate compilation. In *Proc. Principles of Programming Languages*, 1994.

[21] X. Leroy. *The Objective Caml System*, 3.08 edition, 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[22] H. Makholm and J. B. Wells. Type inference, principal typings, and let-polymorphism for first-class mixin modules. In *Proc. International Conference on Functional Programming*, 2005.

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[24] M. Odersky and M. Zenger. Scalable component abstractions. In *Proc. Object Oriented Programming, Systems, Languages, and Applications*, 2005.

[25] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[26] C. V. Russo. Recursive structures for Standard ML. In *Proc. International Conference on Functional Programming*, 2001.

[27] M. Serrano. *Bigloo: A "practical Scheme compiler"*, 2.6e edition, Aug. 2004. `http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html`.

[28] D. Swasey, T. Murphy VII, K. Crary, and R. Harper. A separate compilation extension to Standard ML (working draft). Technical Report CMU-CS-06-104, School of Computer Science, Carnegie Mellon University, January 2006.

[29] D. Syme. Initializing mutually referential abstract objects: The value recursion challenge. In *Proc. Workshop on ML*, 2005.

[30] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, New York, 1997.

[31] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In *Proc. Principles of Programming Languages*, 1999.

[32] J. B. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Proc. European Symposium on Programming Languages and Systems*, 2000.