Units: Cool Modules for HOT Languages

Matthew Flatt Matthias Felleisen
Department of Computer Science*
Rice University
Houston, Texas 77005-1892

Abstract

A module system ought to enable *assembly-line programming* using separate compilation and an expressive linking language. Separate compilation allows programmers to develop parts of a program independently. A linking language gives programmers precise control over the assembly of parts into a whole. This paper presents models of *program units*, MzScheme's module language for assembly-line programming. Units support separate compilation, independent module reuse, cyclic dependencies, hierarchical structuring, and dynamic linking. The models explain how to integrate units with untyped and typed languages such as Scheme and ML.

1 Introduction

Henry Ford had a better idea. His assembly line revolutionized manufacturing with two innovations: *standardized parts* and a *controlled assembly process*. Standardized parts can be independently manufactured, tested, and replaced. A controlled assembly process ensures that parts are assembled reliably. Module systems ought to enable *assembly-line programming*. A module system should provide separate compilation to support the independent development of parts, and it should provide a linking language to give the programmer control over assembling parts.

Existing module systems for HOT (higher-order, typed) languages do not enable assembly-line programming. Some HOT module languages do not provide separate compilation, making it impossible to test and distribute individual modules [3]. In other module languages, *e.g.*, the package languages of Ada, Modula-3, and Java, connections are hard-wired within modules instead of specified in a separate assembly process. Some HOT module systems fail to scale essential features of the core language to modules, which restricts the ways that modules can be defined, *e.g.*, ML[1] does not support mutually recursive procedure and type definitions at the module level. Finally, few HOT module languages handle dynamic program construction and dynamic linking, which are needed for programs with "some assembly required," *e.g.*, web-based applets.

For MzScheme [8], we designed and implemented a language of modules, called *program units*, to support assembly-line programming. In particular, the following properties of our unit language enable the independent development of parts:

- **Encapsulation:** A unit encapsulates a program part, clearly delineating the interface between the unit and all other parts of the program.
- **Separate compilation:** A unit's interface provides enough information for the separate compilation of the unit.

To support the assembly process, the unit language provides the following mechanisms:

- **Individual reuse and replacement:** Individual units are reusable and replaceable. This implies that the connections between units are specified outside the units themselves rather than hard-wired within each unit. In addition, the language supports multiple instances of a unit in different contexts within a program.
- **Hierarchical structuring:** The unit language allows units to be linked together to create a single, larger unit, possibly hiding selected details of the component units in the process.
- **Dynamic linking:** Units support dynamic linking, connecting new and executing code through a well-defined and localized interface.

This paper presents untyped and typed models of units that are suitable for Scheme-like and ML-like languages. For these core languages, scaling essential core features to the module level implies two final properties:

- **Types:** If the core programming language supports static type definitions, units import and export types as well as values.
- **Mutual dependencies:** In whatever manner the core language supports mutually recursive definitions (usually procedure and type definitions), the unit language allows definitions with mutual references across module boundaries.

*This research was partially supported by a NSF Graduate Research Fellowship, NSF grants CCR-9619756, CDA-9713032, and CCR-9708957, and a Texas ATP grant.

[1]ML stands for SML or CAML.

To appear: PLDI – June 17-19 1998, Montreal, Canada

Although our unit language specifies *how* units are defined and linked, there is no specific mechanism for describing *which* units are linked together to form a program. In general, the process of selecting units for a program can be quite complex, as evidenced by elaborate makefiles used to build programs in traditional languages. In our unit language, the programmer writes program-linking programs in the core language itself; units are integrated as first-class values in the core language, and the unit definition and linking forms are core expression forms. The only primitive operations on units are linking and invocation, which preserves separate compilation for individual units, but programmers can exploit the full flexibility of the core language to apply these operations.

Section 2 explains how our unit model relates to existing module languages. Section 3 provides an overview of programming with units, and Section 4 defines the precise syntax, type checking, and semantics of units. Section 5 briefly considers extensions to the typed unit model. The last two sections relate our work to other current research in module languages, and put our work into perspective.

2 Existing Module Languages and Units

The unit model synthesizes ideas from three popular existing module systems: `.o` files, packages, and ML modules. The first represents the traditional view of modules as compilation units. The second extends this view by moving the module language into the programming language. The last gives programmers greater control over how modules are combined into a program.

Traditional languages like C have relied on the filesystem as the language of modules. Programs (makefiles) manipulate `.o` files to select the modules that are linked into a program, and module files are partially linked to create new `.o` or library files. Modern linking systems such as ELF [27] support dynamic linking. However, even the most advanced linking systems rely on a global namespace of function names and module (*i.e.*, file) names. As a result, modules can be linked and invoked only once in a program.

Many modern languages (*e.g.*, Ada 95 [1], Modula-2 [30], Modula-3 [11], Haskell [15], and Java [10]) use *packages*. A package system delineates the boundaries of each module and forces the specification of static dependencies between modules. Since module linking and invocation are clearly separated, packages allow mutually recursive function and type definitions across package boundaries.

The main weakness of a package system is its reliance on a global namespace of packages with hardwired connections among packages. Package systems do not permit the reuse of a single package for multiple invocations in a program or the external selection of connections between packages. (Ada and Modula-3's *generics* allow the former but not the latter.) Packages cannot be merged into a new package that hides parts of the constituent packages. In addition, among the languages with packages, only Java provides a mechanism for dynamic linking. This mechanism is expressed indirectly via the language of class loaders, and is not fully general due to the constraints of a global package namespace.[2]

ML's functor system [22, 24] is the most notable example of a language that lets a programmer describe abstractions over modules and gives a programmer direct control over assembling modules. Unlike package languages, the basic ML module, a *structure*, is not a fragment of unevaluated code. Instead, a structure is a record with fields containing the module's exported values and types. A module with dependencies is defined as a *functor*, a first-order function that consumes a structure and produces a new structure. Functors separate the specification of module dependencies from module linking. Unfortunately, linking by functor application prevents the definition of mutually recursive types or procedures across module boundaries. Worse still, ML provides no mechanism for dynamic linking.

3 Programming with Units

Like a package in Java or Modula-3, a program unit is an unevaluated fragment of code, but there is no global namespace of units. Instead, like an ML functor, a unit describes its import requirements without specifying a particular unit that supplies those imports. The actual linking of the unit is specified externally at a later stage. Unlike in ML, unit linking is specified for groups of units with a graph of connections, which allows mutual recursion across unit boundaries. Furthermore, the result of linking a collection of units is a new (compound) unit that is available for further linking.

This section illustrates the basic design elements of our unit language using an informal, semi-graphical programming language. (The graphical language is currently being implemented for our Scheme programming environment. Programmers will define modules and linking by actually drawing boxes and arrows.) The examples assume a core language with lexical blocks and a sub-language of types. The syntax used for the core language mimics that of ML.

3.1 Defining Units

Figure 1 defines a unit called *Database*. In the graphical notation, a unit is drawn as a box with three sections:

- The top section lists the unit's imported types and values. The *Database* unit imports the type *info* (of kind[3] Ω) for data stored in the database, and the function *error* (of type $\mathsf{str}\rightarrow\mathsf{void}$) for error-handling.

- The middle section contains the unit's definitions and an initialization expression. The latter performs start-up actions for the unit at run-time. The *Database* unit defines the type *db* and the functions *new*, *insert*, and *delete* (plus some other definitions that are not shown). Database entries are keyed by strings, so *Database* initializes a hash table for strings with the expression $\mathit{strTable} := \mathit{makeStringHashTable}()$.

- The bottom section enumerates the unit's exported types and values. The *Database* unit exports the type *db* and the functions *new*, *insert*, and *delete*.

In a statically-typed language, all imported and exported variables have a type, and all imported and exported types have a kind.[3] Imported and defined types can be used in the type expressions for imported and exported values. All exported variables must be defined within the unit, and the type expression for an exported value must use only imported and exported types. In *Database*, both the imported

[2] Java's class system can also be viewed as a kind of module system or as a complement to the package system. Classes suffer the same drawbacks as packages: links, such as a superclass name, are hardwired to a specific class [9].

[3] A kind is a type for a type. Most languages have only one kind, Ω, and do not ask programmers to specify the kind of a type. Some languages (such as ML, Haskell, and Miranda) also provide type constructors or functions on types, which have the kind $\Omega^* \rightarrow \Omega$.

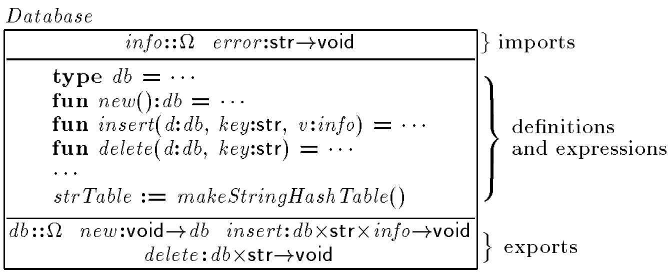

Figure 1: An atomic database unit

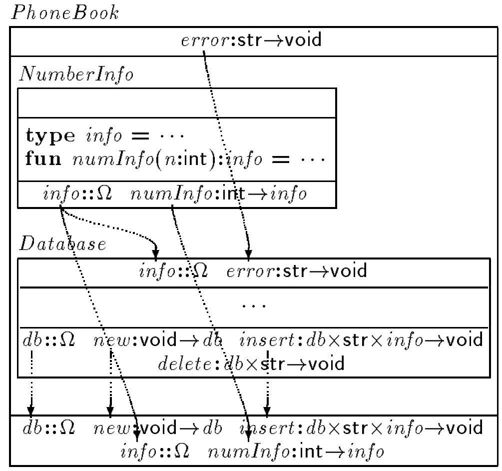

Figure 2: Linking units to form a compound unit

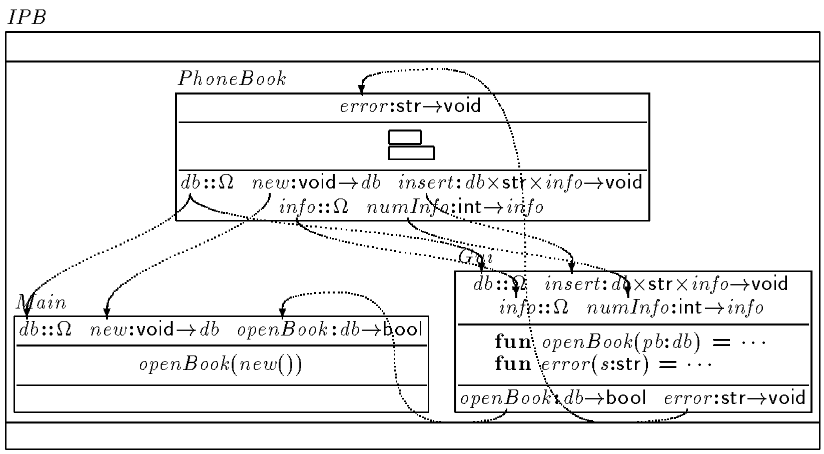

Figure 3: Linking units to define a complete program

type *info* and the exported type *db* are used in the type expression for *insert*: *db*×str×*info*→void.

A unit is specifically *not* a record of values. It encapsulates unevaluated code, much like the `.o` file created by compiling a C++ module. Before a unit's definitions and initialization expression can be evaluated, it must first be linked with other units to resolve all of its imports.

3.2 Linking Units

In the graphical notation, a programmer links units together by drawing arrows to connect the exports of one box with the imports of another. Linking units together creates a compound unit, as illustrated in Figure 2 with the *PhoneBook* unit. This unit links *Database* with *NumberInfo*, a unit that implements the *info* type for phone numbers.

Figure 2 also shows how to link units in stages. The *error* function is not defined by either *Database* or *NumberInfo*, so *PhoneBook* imports *error* and passes the imported value on to *Database*. At the same time, *PhoneBook* hides the *delete* function, but re-exports all of the other values and types from *Database* and *NumberInfo*.

A complete program is a unit without imports. Figure 3 defines a complete interactive phone book program, *IPB* (Interactive Phone Book), which links *PhoneBook* with a graphical interface implementation *Gui*. The *Main*[4] unit contains an initialization expression that creates a database and an associated graphical user interface.

A program unit is analogous to an executable file; *invoking* the unit evaluates the definitions in all of the program's units and then executes their initialization expressions. Thus, invoking *IPB* executes *Main*'s initialization expression, which creates a new phone book database and opens a phone book window. The variables exported by a program are ignored. The result of invoking a program is the value of its last initialization expression—a bool value in *IPB* (assuming *Main*'s expression is evaluated last).[5]

A compound unit's links must satisfy the type require-

[4] The name *Main* is not special.

[5] Our informal graphical notation does not specify the order of units in a compound unit, but the textual notation in Section 4 covers this aspect of the language.

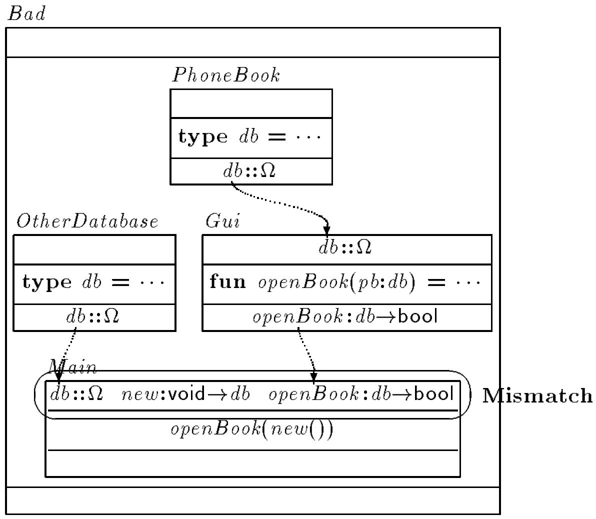

Figure 4: Illegal linking due to a type mismatch

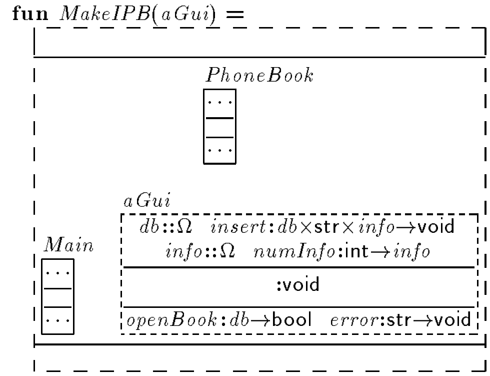

Figure 5: Abstracting over constituent units

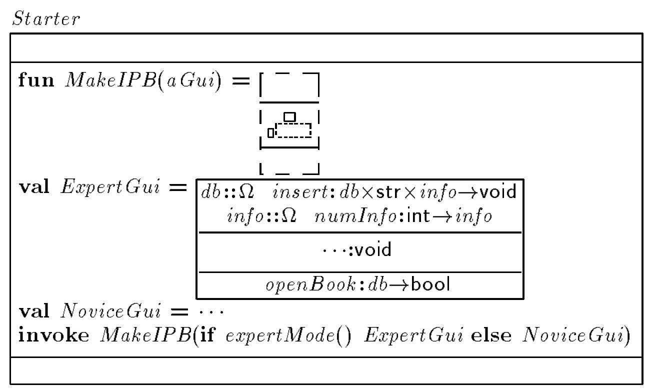

Figure 6: Linking and invoking other programs

ments of the constituent units. For example, in *IPB* (see Figure 3), *Main* imports the type *db* from *PhoneBook* unit and also the function *openBook*:*db*→bool from *Gui*. The two occurrences of *db* must refer to the same type. A type checker can verify this constraint by proving that the two occurrences have the same source in the link graph, which is the *db* exported by *PhoneBook*. In contrast, Figure 4 defines a "program" *Bad* in which inconsistent imports are provided to *Main*. Specifically, *db* and *openBook*:*db*→bool refer to types named *db* that originate from different units. The type checker correctly rejects *Bad* due to this mismatch.

Linking can connect units in a mutually recursive manner. This is illustrated in *IPB* (see Figure 3); links flow both from *PhoneBook* to *Gui* and from *Gui* to *PhoneBook*. Thus, the *insert* function in *PhoneBook* may call *error* in *Gui*, which could in turn call *PhoneBook*'s *insert* again to handle the error.

3.3 Programs that Link and Invoke Other Programs

The *IPB* program relies on a fixed set of constituent units, including a specific unit *Gui* to implement the graphical interface. In general, there may be multiple GUIs that work with the phone book, *e.g.*, separate GUIs for novice and advanced users. Every GUI unit will have the same set of imports and exports, so the linking information required to produce the complete interactive phone book is independent of the specific GUI unit. In short, the *IPB* compound unit could be abstracted with respect to its GUI unit.

If a form for linking units is integrated into the core evaluation language, then the abstraction of *IPB* can be achieved with a core function. Figure 5 defines *MakeIPB*, a function that accepts a GUI unit and returns an interactive phone book unit. The programmer draws a dashed box for *aGui* and *MakeIPB* to indicate that the actual GUI and interactive phone book units are not yet determined. *MakeIPB* can be applied to different GUI implementations to produce different interactive phone book programs.

The type associated with *MakeIPB*'s argument is a unit type, a *signature*, that contains all of the information needed to verify its linkage in *MakeIPB*. In the graphical notation, a signature corresponds to a box with imports, exports, and an initialization expression type, but no definitions or expressions. The signature for *aGui* is defined by its dotted box, with **:void** indicating the type of the initialization expression. Using only this signature, the type system can completely verify the linking in *MakeIPB* and determine the signature of the resulting compound unit.

Figure 6 shows *MakeIPB* as part of a larger program, *Starter*, that selects a GUI unit and links together a complete interactive phone book program. Once *MakeIPB* returns a program unit, *Starter* launches the constructed program with the special **invoke** form, which takes a program unit and executes it.

3.4 Dynamic Linking

The **invoke** form also works on units that are not complete programs. In this case, the unit's imports are explicitly satisfied by types and values from the invoking program. This generalized form of invocation implements *dynamic linking*. For example, the phone book program can exploit dynamic linking to support third-party "plug-in" extensions that load phone numbers from a foreign source. Each such loader extension is implemented as a unit that is dynamically retrieved from an archive and then linked with the phone book

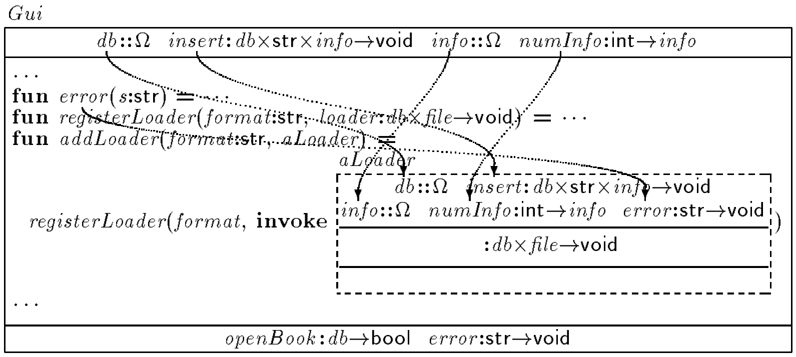

Figure 7: Dynamic linking with **invoke**

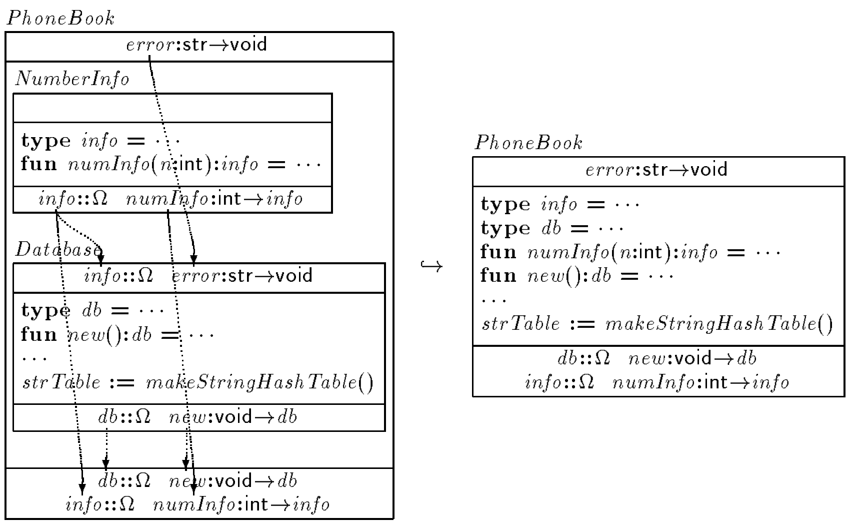

Figure 8: Graphical reduction rule for a compound unit

program.[6] Now, the user of the phone book can install a loader extension at run-time via interactive dialogues.

Figure 7 defines a *Gui* unit that supports loader extensions. The function *addLoader* consumes a loader extension as a unit and dynamically links it into the program using **invoke**. The extension unit imports types and functions that enable it to modify the phone book database. These imports are satisfied in the **invoke** expression with types and variables that were originally imported into *Gui*, plus the *error* function defined within *Gui*. The result of invoking the extension unit is the value of the unit's initialization expression, which is required (via signatures) to be a function of type $db \times file \rightarrow$void. This function is then installed into the GUI's table of loader functions.

[6]The core language must provide a syntactic form that retrieves a unit value from an archive, such as the Internet, and checks that the unit satisfies a particular signature. This type-checking must be performed in the correct context to ensure that dynamic linking is type-safe. Java's dynamic class loading is broken because it checks types in a type environment that may differ from the environment where the class is used [26].

4 The Structure and Interpretation of Units

In this section we develop a semantic and type-theoretic account of the unit language design in three stages. We start in Section 4.1 with units as an extension of a dynamically typed language (like Scheme) to introduce the basic syntax and semantics of units. In Section 4.2, we enrich this language with definitions for constructed types (like classes in Java or datatypes in ML). Finally, in Section 4.3 we consider arbitrary type definitions (like type equations in ML). For all three sections, we only consider those parts of the core language that are immediately relevant to units.

The rigorous description of the unit language, including its type structures and semantics, relies on well-known type checking and rewriting techniques for Scheme and ML [6, 13, 31]. In the rewriting model of evaluation, the set of program expressions is partitioned into a set of values and a set of non-values. Evaluation is the process of rewriting a non-value expression within a program to an equivalent expression, repeating this process until the whole program is rewritten to a value. For example, an atomic unit expression—represented in the graphical language by a box

e	=	**unit** *imports exports definitions e*
	|	**compound** *imports exports* **link** *e link* **and** *e link*
	|	**invoke** *e* **with** *invoke-link*
	|	*e* ; *e* | *letrec-expr*
	|	... other core forms ...
letrec-expr	=	**letrec** *value-defn** **in** *e*
imports	=	**import** *value-var-decl**
exports	=	**export** *value-var-decl**
definitions	=	*value-defn**
value-defn	=	**val** *value-var-decl* **=** *e*
link	=	**with** *value-var-decl** **provides** *value-var-decl**
invoke-link	=	*value-invoke-link**
value-invoke-link	=	*value-var-decl* **=** *e*
value-var-decl	=	*x*
x	=	variable

Figure 9: Syntax for $\text{U}\textsc{nit}_\mathsf{d}$ (dynamically typed)

containing text code—is a value, while a compound unit expression—a box containing linked boxes—is not a value.

A compound unit expression with known constituents can be re-written to an equivalent unit expression by merging the text of its constituent units, as demonstrated in Figure 8. Invocation for a unit is similar: an **invoke** expression is rewritten by extracting the invoked unit's definitions and initialization expression, and then replacing references to imported variables with values. Otherwise, the standard rules for functions, assignments, and exceptions apply.

4.1 Dynamically Typed Units

Figure 9 defines the syntax of $\text{U}\textsc{nit}_\mathsf{d}$, an extension of a dynamically typed core language. The core language must provide two forms that are used in the process of linking and invoking: an expression sequence form (";") and a **letrec** form for lexical blocks containing mutually recursive definitions. The core language is extended with three unit-specific forms:

- a **unit** form for creating units;
- a **compound** form for linking units; and
- an **invoke** form for invoking units.

4.1.1 The unit Form

The **unit** form consists of a set of import and export declarations followed by internal definitions and an initialization expression:

$$\begin{array}{l} \textbf{unit import } x_i \cdots \textbf{ export } x_e \cdots \\ \quad \textbf{val } x = e_v \quad \cdots \\ \quad e \end{array}$$

The imported variables x_i are bound in the definition and initialization expressions. The exported variables x_e must be defined within the unit. The scope of a defined variable includes all of the definition expressions e_v in the unit as well as the initialization expression e.

In each definition **val** $x = e_v$, the expression e_v must be *valuable* in the sense of Harper and Stone [14], with the restriction that imported and defined variable names are not considered valuable. The intent of this restriction is that evaluating the expression terminates, does not incur any computational effects (divergence, printing, *etc.*), and does not refer to variables whose values may still be undetermined (due to an ordering of the mutually recursive definitions).[7]

A **unit** expression is a first-class value, just like a number or an object in Java. There are only two operations on units: linking and invoking. No operation can "look inside" a unit value to extract any information about its definitions or initialization expression. In particular, since a unit does not contain any values (only unevaluated expressions), there is no "dot notation" for externally accessing values from a unit (as in ML) and there are no "instantiated units" (approximating an ML structure) that contain the values of unit expressions.

To simplify the presentation, $\text{U}\textsc{nit}_\mathsf{d}$ does not allow α-renaming for a unit's imported and exported variables. In MzScheme's units, imported and exported variables have separate internal (binding) and external (linking) names, and the internal names within a unit can be α-renamed.

4.1.2 The compound Form

The **compound** form links two constituent units together into a new unit:

$$\begin{array}{l} \textbf{compound import } x_i \cdots \textbf{ export } x_e \cdots \\ \quad \textbf{link } e_1 \textbf{ with } x_{i1} \textbf{ provides } x_{e1} \\ \quad \textbf{and } e_2 \textbf{ with } x_{i2} \textbf{ provides } x_{e2} \end{array}$$

The constituent units are determined by two subexpressions: e_1 and e_2. Along with each expression, the variables that the unit is expected to import are listed following the **with** keyword, and the variables that the unit is expected to export are listed following the **provides** keyword.

Variables are linked within **compound** by name. Thus, the set of variables x_{i1} linked into the first unit must be a subset of $x_i \cup x_{e2}$. Similarly, x_{i2} must be a subset of $x_i \cup x_{e1}$. Finally, the set variables x_e exported by the compound unit must be a subset of $x_{e1} \cup x_{e2}$.

A **compound** unit expression is not a value. It evaluates to a unit value that is indistinguishable from an atomic unit. This unit's initialization expression is the sequence of the first constituent unit's initialization expression followed by the the second constituent unit's.

Once again, MzScheme's syntax is less restrictive than $\text{U}\textsc{nit}_\mathsf{d}$'s. In MzScheme, the **compound** form links any number of units together at once (a simple generalization of $\text{U}\textsc{nit}_\mathsf{d}$'s two-unit form), and links imports and exports via source and destination name pairs, rather than requiring the same name at both ends of a linkage.

4.1.3 The invoke Form

The **invoke** form evaluates its first subexpression to a unit and invokes it:

$$\textbf{invoke } e \textbf{ with } x_i = e_i \cdots$$

If the unit requires any imported values, they must be provided through $x_i = e_i$ declarations, which associate values e_i with names x_i for the unit's imports. An **invoke** expression evaluates to the invoked unit's initialization expression.

[7] This last restriction simplifies the presentation of the formal semantics, but it can be lifted for an implementation, as in MzScheme, where accessing an undefined variable returns a default value or signals a run-time error.

$$\frac{\overline{x}\ \text{distinct}\qquad \Gamma\vdash e_u\qquad \Gamma\vdash\overline{e}}{\Gamma\vdash\textbf{invoke}\ e_u\ \textbf{with}\ \overline{x=e}}$$

$$\frac{\begin{array}{c}\overline{x_i}\cup\overline{x}\ \text{distinct}\qquad \overline{x_e}\subseteq\overline{x}\\ \Gamma,\overline{x},\overline{x_i}\vdash\overline{e}\qquad \Gamma,\overline{x},\overline{x_i}\vdash e_b\end{array}}{\begin{array}{l}\Gamma\vdash\textbf{unit import}\ \overline{x_i}\ \textbf{export}\ \overline{x_e}\\ \qquad\qquad \overline{\textbf{val}\ x=e}\ \textbf{in}\ e_b\end{array}}$$

$$\frac{\begin{array}{c}\overline{x_i}\cup\overline{x_{p1}}\cup\overline{x_{p2}}\ \text{distinct}\qquad \overline{x_e}\ \text{distinct}\\ \overline{x_{w1}}\subseteq\overline{x_i}\cup\overline{x_{p2}}\qquad \overline{x_{w2}}\subseteq\overline{x_i}\cup\overline{x_{p1}}\qquad \overline{x_e}\subseteq\overline{x_{p1}}\cup\overline{x_{p2}}\qquad \Gamma\vdash e_1\qquad \Gamma\vdash e_2\end{array}}{\begin{array}{l}\Gamma\vdash\textbf{compound import}\ \overline{x_i}\ \textbf{export}\ \overline{x_e}\\ \qquad\qquad \textbf{link}\ e_1\ \textbf{with}\ \overline{x_{w1}}\ \textbf{provides}\ \overline{x_{p1}}\\ \qquad\qquad \textbf{and}\ e_2\ \textbf{with}\ \overline{x_{w2}}\ \textbf{provides}\ \overline{x_{p2}}\end{array}}$$

The notation $\overline{x}$ indicates either a set or a sequence of variables x, depending on the context. The notation $\overline{\textbf{val}\ x=e}$ indicates the sequence $\textbf{val}\ x=e$ where each x is taken from the sequence $\overline{x}$ with a corresponding e from the sequence $\overline{e}$.

Figure 10: Checking the form of UNIT_d expressions

$$\begin{array}{l}\textbf{invoke}\ (\textbf{unit import}\ \overline{x_i}\ \textbf{export}\ \overline{x_e}\\ \qquad\qquad \overline{\textbf{val}\ x=e}\ \textbf{in}\ e_b)\\ \qquad \textbf{with}\ \overline{x_w=v_w}\end{array}\quad\hookrightarrow\quad [\overline{v_w/x_w}](\textbf{letrec}\ \overline{\textbf{val}\ x=e}\ \textbf{in}\ e_b)\ \text{ if } \overline{x_i}\subseteq\overline{x_w}$$

$$\begin{array}{l}\textbf{compound import}\ \overline{x_i}\ \textbf{export}\ \overline{x_e}\\ \qquad \textbf{link}\ (\textbf{unit import}\ \overline{x_{i1}}\ \textbf{export}\ \overline{x_{e1}}\\ \qquad\qquad\quad \overline{\textbf{val}\ x_1=e_1}\ \textbf{in}\ e_{b1})\quad \textbf{with}\ \overline{x_{w1}}\ \textbf{provides}\ \overline{x_{p1}}\\ \qquad \textbf{and}\ (\textbf{unit import}\ \overline{x_{i2}}\ \textbf{export}\ \overline{x_{e2}}\\ \qquad\qquad\quad \overline{\textbf{val}\ x_2=e_2}\ \textbf{in}\ e_{b2})\quad \textbf{with}\ \overline{x_{w2}}\ \textbf{provides}\ \overline{x_{p2}}\\ \text{if } \overline{x_1}\cup\overline{x_2}\cup\overline{x_i}\ \text{distinct},\ \overline{x_{i1}}\subseteq\overline{x_{w1}},\ \overline{x_{p1}}\subseteq\overline{x_{e1}},\ \overline{x_{i2}}\subseteq\overline{x_{w2}},\ \text{and}\ \overline{x_{p2}}\subseteq\overline{x_{e2}}\end{array}\quad\hookrightarrow\quad\begin{array}{l}\textbf{unit import}\ \overline{x_i}\\ \quad \textbf{export}\ \overline{x_e}\\ \quad \overline{\textbf{val}\ x_1=e_1}\\ \quad \overline{\textbf{val}\ x_2=e_2}\\ \quad \textbf{in}\ e_{b1}\ ;\ e_{b2}\end{array}$$

Figure 11: Reducing UNIT_d expressions

4.1.4 UNIT_d Context-sensitive Checking

The rules in Figure 10 specify the context-sensitive properties that were informally described in the previous section. The checks ensure that a variable is not multiply defined, imported, or exported, that all exported variables are defined, and that the **link** clause of a **compound** expression is locally consistent.

4.1.5 UNIT_d Evaluation

The unit-specific reduction rules for UNIT_d, defined in Figure 11, generalize the graphical example in Figure 8. The rules extend those for Scheme [6] and resemble equations in the higher-order module calculus of Harper, Mitchell, and Moggi [13]. The first rule specifies that an **invoke** expression reduces to a **letrec** expression containing the invoked unit's definitions and initialization expression. In this **letrec** expression, imported variables are replaced by values. The set of variables supplied by **invoke**'s **with** clause must cover the set of the imports required by the unit; otherwise, a run-time error is signalled.

The second rule defines how a **compound** expression combines two units: their definitions are merged and their initialization expressions are sequenced. The **compound** rule requires that the constituent units provide at least the expected exports (according to the **provides** clauses) and need no more than the expected imports (according to the **with** clauses). Also, all bindings introduced by definitions in the two units must be appropriately α-renamed to avoid collisions.

```
unit import even
     export odd
  val odd = fn 0 ⇒ false
               | n ⇒ even (n-1)
  odd 13

⟹

fn (evencell, oddcell) ⇒
 (oddcell := (fn 0 ⇒ false
                 | n ⇒ (!evencell) (n-1));
  fn () ⇒ (!oddcell) 13)
```

Figure 12: An example of UNIT_d compilation

4.1.6 UNIT_d Implementation

In MzScheme's implementation of UNIT_d, units are compiled by transforming them into functions. The unit's imported and exported variables are implemented as first-class reference cells that are externally created and passed to the function when the unit is invoked. The function is responsible for filling the export cells with exported values and for remembering the import cells for accessing imports later. The return value of the function is a closure that evaluates the unit's initialization expression. Figure 12 illustrates this transformation on an atomic unit.

A compound unit is also compiled to a function. The function encapsulates a list of constituent units and a closure that propagates import and export cells to the constituent units, creating new cells to implement variables in

$$
\begin{array}{rcl}
\textit{letrec-expr} & = & \textbf{letrec}\ \textit{type-defn}*\ \textit{value-defn}*\ \textbf{in}\ e \\
\textit{imports} & = & \textbf{import}\ \textit{type-var-decl}*\ \textit{value-var-decl}* \\
\textit{exports} & = & \textbf{export}\ \textit{type-var-decl}*\ \textit{value-var-decl}* \\
\textit{definitions} & = & \textit{datatype-defn}*\ \textit{value-defn}* \\
\textit{datatype-defn} & = & \textbf{type}\ t = x{,}x\ \tau \mid x{,}x\ \tau \rhd x \\
\textit{link} & = & \textbf{with}\ \textit{type-var-decl}*\ \textit{value-var-decl}* \\
 & & \quad \textbf{provides}\ \textit{type-var-decl}*\ \textit{value-var-decl}* \\
\textit{invoke-link} & = & \textit{type-invoke-link}*\ \textit{value-invoke-link}* \\
\textit{type-invoke-link} & = & \textit{type-var-decl} = \tau \\
\textit{type-var-decl} & = & t :: \kappa \\
\textit{value-var-decl} & = & x : \tau \\
 & & \\
\tau, \sigma & = & t \mid \tau \rightarrow \tau \mid \textbf{signature} \\
\textbf{signature} & = & \textbf{sig}\ \textit{imports}\ \textit{exports}\ \tau \\
 & & \\
t & = & \text{type variable} \\
\kappa & = & \text{type kind}
\end{array}
$$

Figure 13: Syntax for UNIT_{c} (constructed types)

the constituents that are hidden by the compound unit.

The transformed units have the same code-sharing properties as traditional shared libraries. The definition and initialization expressions of a unit are compiled in the body of the function produced by its transformation, and this one function is used for all instances of the unit. Thus, there exists a single copy of the definition and initialization code regardless of how many times the unit is linked or invoked.[8]

4.2 Units with Constructed Types

Figure 13 extends the language in Figure 9 for a statically typed language with programmer-defined constructed types, such as ML datatypes. In the new language, UNIT_{c}, the imports and exports of a **unit** expression include type variables as well as value variables. All type variables have a kind[9] and all value variables have a type. The **compound** and **invoke** expressions are extended in the natural way to handle imported and exported types.

The definition section of a **unit** expression contains both type and value definitions. Type definitions are similar to ML **datatype** definitions, but for simplicity, every type defined in UNIT_{c} has exactly two variants. Type definitions have the form **type** $t = x_{\mathsf{cl}}, x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}}, x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \rhd x_{\mathsf{t}}$. Instances of the first variant are constructed with the x_{cl} function, which takes a value of type τ_{l} and constructs a value of type t. They are deconstructed with x_{dl}. Instances of the second variant are constructed with x_{cr} given a value of type τ_{r} and deconstructed with x_{dr}. Applying a deconstructor to the wrong variant signals a run-time error. To distinguish variants, the x_{t} function returns true for an instance of the first variant and false for an instance of the second.

The τ_{l} and τ_{r} type expressions can refer to t or other type variables to form recursive or mutually recursive type definitions. We assume that the core language for UNIT_{c} provides a **letrec** form for mutually recursive procedure and datatype definitions.

The type of a **unit** expression is a signature of the form **sig** *imports exports* τ where *imports* specifies the kinds and types of a unit's imports and *exports* describes the kinds and types of its exports. In a **sig** form, as in a **unit** form, types in either *imports* or *exports* can be used in the type expressions within the signature. The type expression τ is the type of the unit's initialization expression, which cannot depend on type variables listed in *exports*.

[8]Our native code compiler transforms a **unit** expression to a shared library that is managed by the operating system.

[9]Although the only kind in this language is Ω, we declare kinds explicitly in anticipation of future work that handles type constructors and polymorphism.

$$
\frac{\begin{array}{c}
\tau_{b1} \le \tau_{b2} \quad \overline{t_{i1}{::}\kappa_{i1}} \subseteq \overline{t_{i2}{::}\kappa_{i2}} \quad \overline{t_{e1}{::}\kappa_{e1}} \supseteq \overline{t_{e2}{::}\kappa_{e2}} \\
\forall x_{i1}{:}\tau_{i1} \in \overline{x_{i1}{:}\tau_{i1}}, \exists x_{i1}{:}\tau_{i2} \in \overline{x_{i2}{:}\tau_{i2}} : \tau_{i2} \le \tau_{i1} \\
\forall x_{e2}{:}\tau_{e2} \in \overline{x_{e2}{:}\tau_{e2}}, \exists x_{e2}{:}\tau_{e1} \in \overline{x_{e1}{:}\tau_{e1}} : \tau_{e1} \le \tau_{e2}
\end{array}}{\mathsf{sig}[\mathit{i1}, \mathit{e1}, \mathit{b1}] \le \mathsf{sig}[\mathit{i2}, \mathit{e2}, \mathit{b2}]}
$$

$$
\frac{\Gamma \vdash e : \tau' \quad \tau' \le \tau}{\Gamma \vdash_{\mathsf{s}} e : \tau}
$$

Figure 14: Subtyping and subsumption in UNIT_{c} signatures

4.2.1 UNIT_{c} Type Checking

For economy, we introduce the following unusual abbreviation, which summarizes the content of a signature with the indices used on names:

$$
\begin{array}{rl}
\mathsf{sig}[i, e, b] \equiv \textbf{sig} & \textbf{import}\ \overline{t_i{::}\kappa_i}\ \overline{x_i{:}\tau_i} \\
 & \textbf{export}\ \overline{t_e{::}\kappa_e}\ \overline{x_e{:}\tau_e} \\
 & \tau_b
\end{array}
$$

Signatures have a subtype relation to allow the use of specialized units in place of more general units. As defined in Figure 14, a specific signature t_{s} is a subtype of a more general signature t_{g} ($t_{\mathsf{s}} \le t_{\mathsf{g}}$) if:

1. the type of the initialization expression in t_{s} is a subtype of the one in t_{g};
2. t_{s} has fewer imports and more exports than t_{g};
3. for each imported variable in t_{s}, its type in t_{g} is a subtype of its type in t_{s}; and
4. for each exported variable in t_{g}, its type in t_{s} is a subtype of its type in t_{g}.

The typing rules for UNIT_{c} are shown in Figure 15. These rules are typed extensions of the rules from Section 4.1.4. The special judgement $\vdash_{\mathsf{s}}$ is used when subsumption is allowed on an expression's type. Subsumption is used carefully so that type checking is deterministic. For example, subsumption is not allowed for the e_u expression in the **invoke** rule because the initialization expression type τ_b in e_u's signature supplies the type of the entire **invoke** expression.

The first typing rule checks the well-formedness of a signature. Each of the type expressions in a signature must be well-formed in an environment containing the signature's imported and exported type variables, and the type expression for the initialization expression must not refer to any of the exported type variables.

The second rule checks **invoke** expressions, first ensuring that the **with** clause is well-formed. The first expression in an **invoke** form must have a signature type whose imports match the **with** clause. The exports in the signature are ignored. The initialization expression's type in this signature is the type of the complete **invoke** expression.

$$\frac{\begin{array}{c}\Gamma' = \Gamma, \overrightarrow{t_i{::}\kappa_i}, \overrightarrow{t_e{::}\kappa_e} \quad FTV(\tau_b) \cap \overrightarrow{t_e} = \emptyset \\ \Gamma' \vdash \tau_i :: \kappa_i \quad \Gamma' \vdash \tau_e :: \kappa_e \quad \Gamma' \vdash \tau_b :: \Omega\end{array}}{\Gamma \vdash \mathsf{sig}[i, e, b] :: \Omega}
\qquad
\frac{\begin{array}{c}\vec{t} \cup \overrightarrow{x} \text{ distinct} \quad \Gamma \vdash \tau_b :: \Omega \quad \Gamma \vdash \overline{\sigma} :: \overline{\kappa} \\ \Gamma \vdash_{\mathsf{s}} \overline{e} : \overline{\tau} \quad \Gamma \vdash e_u : \mathsf{sig}[i, e, b] \\ \mathsf{sig}[i, e, b] \leq \textbf{sig import } \overrightarrow{t{::}\kappa}\ \overrightarrow{x{:}\tau} \textbf{ export } \emptyset\ \tau_b\end{array}}{\Gamma \vdash \textbf{invoke } e_u \textbf{ with } \overrightarrow{t{::}\kappa = \sigma}\ \overrightarrow{x{:}\tau = e} : \tau_b}$$

$$\frac{\begin{array}{c}\overrightarrow{t_i} \cup \vec{t} \cup \overrightarrow{x_i} \cup \overrightarrow{x_{\mathsf{cl}}} \cup \overrightarrow{x_{\mathsf{dl}}} \cup \overrightarrow{x_{\mathsf{cr}}} \cup \overrightarrow{x_{\mathsf{dr}}} \cup \overrightarrow{x_{\mathsf{t}}} \cup \overrightarrow{x} \text{ distinct} \quad \overrightarrow{t_e{::}\kappa_e} \cup \overrightarrow{x_e{:}\tau_e} \subseteq \overrightarrow{t{::}\Omega} \cup \overrightarrow{x{:}\tau} \cup \overrightarrow{x_{\mathsf{cl}}{:}\tau_{\mathsf{l}} \to t} \cup \overrightarrow{x_{\mathsf{dl}}{:}t \to \tau_{\mathsf{l}}} \cup \overrightarrow{x_{\mathsf{cr}}{:}\tau_{\mathsf{r}} \to t} \cup \overrightarrow{x_{\mathsf{dr}}{:}t \to \tau_{\mathsf{r}}} \cup \overrightarrow{x_{\mathsf{t}}{:}t \to \mathsf{bool}} \\ \Gamma \vdash \mathsf{sig}[i, e, b] :: \Omega \qquad \Gamma' = \Gamma, \overrightarrow{t_i{::}\kappa_i}, \overrightarrow{t{::}\Omega} \qquad \Gamma' \vdash \overline{\tau_{\mathsf{l}}} :: \Omega \quad \Gamma' \vdash \overline{\tau_{\mathsf{r}}} :: \Omega \quad \Gamma' \vdash \overline{\tau} :: \Omega \\ \Gamma'' = \Gamma', \overrightarrow{x_i{:}\tau_i}, \overrightarrow{x_{\mathsf{cl}}{:}\tau_{\mathsf{l}} \to t}, \overrightarrow{x_{\mathsf{dl}}{:}t \to \tau_{\mathsf{l}}}, \overrightarrow{x_{\mathsf{cr}}{:}\tau_{\mathsf{r}} \to t}, \overrightarrow{x_{\mathsf{dr}}{:}t \to \tau_{\mathsf{r}}}, \overrightarrow{x_{\mathsf{t}}{:}t \to \mathsf{bool}} \\ \Gamma'' \vdash_{\mathsf{s}} \overline{e} : \overline{\tau} \quad \Gamma'' \vdash e_b : \tau_b\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{unit import } \overrightarrow{t_i{::}\kappa_i}\ \overrightarrow{x_i{:}\tau_i} \quad \textbf{export } \overrightarrow{t_e{::}\kappa_e}\ \overrightarrow{x_e{:}\tau_e} \\ \qquad \overline{\textbf{type } t = x_{\mathsf{cl}}, x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}}, x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \rhd x_{\mathsf{t}}} \\ \qquad \overline{\textbf{val } x{:}\tau = e} \textbf{ in } e_b \\ \quad : \mathsf{sig}[i, e, b]\end{array}}$$

$$\frac{\begin{array}{c}\overrightarrow{t_i} \cup \overrightarrow{t_{p1}} \cup \overrightarrow{t_{p2}} \cup \overrightarrow{x_i} \cup \overrightarrow{x_{p1}} \cup \overrightarrow{x_{p2}} \text{ distinct} \quad \overrightarrow{t_e} \cup \overrightarrow{x_e} \text{ distinct} \\ \overrightarrow{t_{w1}{::}\kappa_{w1}} \cup \overrightarrow{x_{w1}{:}\tau_{w1}} \subseteq \overrightarrow{t_i{::}\kappa_i} \cup \overrightarrow{t_{p2}{::}\kappa_{p2}} \cup \overrightarrow{x_i{:}\tau_i} \cup \overrightarrow{x_{p2}{:}\tau_{p2}} \\ \overrightarrow{t_{w2}{::}\kappa_{w2}} \cup \overrightarrow{x_{w2}{:}\tau_{w2}} \subseteq \overrightarrow{t_i{::}\kappa_i} \cup \overrightarrow{t_{p1}{::}\kappa_{p1}} \cup \overrightarrow{x_i{:}\tau_i} \cup \overrightarrow{x_{p1}{:}\tau_{p1}} \\ \overrightarrow{t_e{::}\kappa_e} \cup \overrightarrow{x_e{:}\tau_e} \subseteq \overrightarrow{t_{p1}{::}\kappa_{p1}} \cup \overrightarrow{t_{p2}{::}\kappa_{p2}} \cup \overrightarrow{x_{p1}{:}\tau_{p1}} \cup \overrightarrow{x_{p2}{:}\tau_{p2}} \\ \Gamma \vdash e_1 : \mathsf{sig}[i1, e1, b1] \quad \Gamma \vdash e_2 : \mathsf{sig}[i2, e2, b2] \\ \Gamma \vdash \mathsf{sig}[w1, p1, b1] :: \Omega \quad \Gamma \vdash \mathsf{sig}[w2, p2, b2] :: \Omega \\ \mathsf{sig}[i1, e1, b1] \leq \mathsf{sig}[w1, p1, b1] \quad \mathsf{sig}[i2, e2, b2] \leq \mathsf{sig}[w2, p2, b2] \\ \Gamma \vdash \mathsf{sig}[i, e, b2] :: \Omega\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{compound import } \overrightarrow{t_i{::}\kappa_i}\ \overrightarrow{x_i{:}\tau_i} \textbf{ export } \overrightarrow{t_e{::}\kappa_e}\ \overrightarrow{x_e{:}\tau_e} \\ \qquad\qquad \textbf{link } e_1 \textbf{ with } \overrightarrow{t_{w1}{::}\kappa_{w1}}\ \overrightarrow{x_{w1}{:}\tau_{w1}} \textbf{ provides } \overrightarrow{t_{p1}{::}\kappa_{p1}}\ \overrightarrow{x_{p1}{:}\tau_{p1}} \\ \qquad\qquad \textbf{and } e_2 \textbf{ with } \overrightarrow{t_{w2}{::}\kappa_{w2}}\ \overrightarrow{x_{w2}{:}\tau_{w2}} \textbf{ provides } \overrightarrow{t_{p2}{::}\kappa_{p2}}\ \overrightarrow{x_{p2}{:}\tau_{p2}} \\ \quad : \mathsf{sig}[i, e, b2]\end{array}}$$

Figure 15: Type checking for UNIT$_\mathsf{c}$

The third rule determines the signature of a **unit** expression. The first line of antecedents contains simple context-sensitive syntax checks as in UNIT$_\mathsf{d}$. In the second line, all of the type expressions in the unit are checked in an environment that is extended with the unit's imported and defined types. Once the type expressions are validated, the environment is extended again, this time with the types for imported and defined variables. Finally, the types of all definition expressions are verified. Subsumption is allowed for all expressions except the initialization expression, which helps determine the overall signature for the unit.

The fourth and final rule verifies the linking in a **compound** expression and determines its signature. The first four lines of antecedents are simple context-sensitive syntax checks. Then, the constituent unit expressions are checked, obtaining signatures for the constituent units. Each of these signatures must approximate a signature derived from the **with** and **provides** clauses in the corresponding linking line. Finally, the signature of the compound unit is defined by the **import** and **export** clauses and the type of the initialization expression in the second constituent unit.

4.2.2 UNIT$_\mathsf{c}$ Evaluation

The reduction rules for UNIT$_\mathsf{c}$ are nearly the same as the rules for UNIT$_\mathsf{d}$ in Figure 11. The only difference for UNIT$_\mathsf{c}$ is that the **invoke** and **compound** reductions propagate **type** definitions as well as **val** definitions.

4.2.3 Type Soundness

If we were to combine UNIT$_\mathsf{c}$ with the monomorphic subset of ML, we could prove a Milner-style type soundness theorem using a subject reduction argument along the lines of Wright and Felleisen [31].

4.2.4 UNIT$_\mathsf{c}$ Implementation

Closed units in UNIT$_\mathsf{c}$ can be compiled separately in the same way as closed functors in ML. When compiling a unit, imported types are obviously not yet determined and thus have unknown representations. Hence, expressions involving imported types must be compiled like polymorphic functions in ML [19, 28]. Otherwise, the restrictions implied by a unit's interface allow inter-procedural optimizations within the unit (such as inlining, specialization, and dead-code elimination). Furthermore, since a compound unit is equivalent to a simple unit that merges its constituent units, *intra*-unit optimization techniques naturally extend to *inter*-unit optimizations when a **compound** expression has known constituent units.

4.3 Units with Type Dependencies and Equations

UNIT$_\mathsf{c}$ supports a core language where each type is associated with a distinct and independent constructor, but this view of types is too strict for many languages. For example, in Java, the constructor that instantiates a class depends on the constructor for the superclass. Other languages, such as

$$
\begin{array}{rcl}
\textit{definitions} & = & \textit{type-defn}^* \ \textit{datatype-defn}^* \ \textit{value-defn}^* \\
\textit{type-defn} & = & \textbf{type}\ t :: \kappa = \sigma \\
\textbf{signature} & = & \textbf{sig}\ \textit{imports exports} \\
 & & \quad \textbf{depends}\ \textit{dependency}^*\ \tau \\
\textit{dependency} & = & t \rightsquigarrow t
\end{array}
$$

Figure 16: Syntax for $\text{U}\scriptstyle{\text{NIT}}_\mathsf{e}$ (type equations)

$$
\frac{\begin{array}{c}\tau_{b1} \leq \tau_{b2} \quad \overline{t_{i1}::\kappa_{i1}} \subseteq \overline{t_{i2}::\kappa_{i2}} \quad \overline{t_{e1}::\kappa_{e1}} \supseteq \overline{t_{e2}::\kappa_{e2}} \\ \overline{t_{de1} \rightsquigarrow t_{di1}} \subseteq \overline{t_{de2} \rightsquigarrow t_{di2}} \\ \forall x_{i1}{:}\tau_{i1} \in \overline{x_{i1}{:}\tau_{i1}}, \exists x_{i1}{:}\tau_{i2} \in \overline{x_{i2}{:}\tau_{i2}} : \tau_{i2} \leq \tau_{i1} \\ \forall x_{e2}{:}\tau_{e2} \in \overline{x_{e2}{:}\tau_{e2}}, \exists x_{e2}{:}\tau_{e1} \in \overline{x_{e1}{:}\tau_{e1}} : \tau_{e1} \leq \tau_{e2}\end{array}}{\mathsf{sig}[i1, e1, di1, de1, b1] \leq \mathsf{sig}[i2, e2, di2, de2, b2]}
$$

Figure 17: Subtyping in $\text{U}\scriptstyle{\text{NIT}}_\mathsf{e}$ signatures

ML, support type equations that introduce new types without explicit constructors; a type equation of the form **type** $t = \tau$ defines the type variable t as an abbreviation for the type expression τ.

Naively mixing units with type dependencies and equations leads to problems. Since two units can contain mutually recursive definitions, linking units with type dependencies may result in cyclic definitions, which core languages like ML and Java do not support. To prevent these cycles, signatures must include information about dependencies between imported and exported types. The dependency information can be used to verify that cyclic definitions are not created in linking expressions.

UNIT$_\mathsf{e}$ extends UNIT$_\mathsf{c}$ with type dependencies and equations. Figure 16 defines syntax extensions for UNIT$_\mathsf{e}$, including a new signature form that contains a **depends** clause. The dependency declaration $t_e \rightsquigarrow t_i$ means that an exported type t_e depends on an imported type t_i. When two units are linked with a **compound** expression, tracing the set of dependencies can ensure that linking does not create a cyclic type definition. Also, the signature for a **compound** expression propagates dependency information for types imported into and exported from the compound unit.

4.3.1 UNIT$_\mathsf{e}$ Type Checking

The following abbreviation expresses a UNIT$_\mathsf{e}$ signature:

$$
\begin{array}{rl}
\mathsf{sig}[i, e, di, de, b] \equiv \textbf{sig} & \textbf{import}\ \overline{t_i{::}\kappa_i}\ \overline{x_i{:}\tau_i} \\
 & \textbf{export}\ \overline{t_e{::}\kappa_e}\ \overline{x_e{:}\tau_e} \\
 & \textbf{depends}\ \overline{t_{de} \rightsquigarrow t_{di}} \\
 & \tau_b
\end{array}
$$

The subtyping rule in Figure 17 accounts for the new dependency declarations. Specifically, a signature is more specific than another if it declares more dependencies.

The type checking rules for UNIT$_\mathsf{e}$ are defined in Figure 19. To calculate type dependencies, the type checking rules employ the "depends on" relation, $\propto_D$. It associates a type expression with each of the type variables it references from the set of type equations D:

$$
\begin{array}{l}
\tau \propto_D t \text{ iff } t \in FTV(\tau) \\
\qquad\qquad \text{or } (\exists \langle t' = \tau' \rangle \in D : t' \in FTV(\tau) \text{ and } \tau' \propto_D t)
\end{array}
$$

$FTV(\tau)$ denotes the set of type variables in τ that are not bound by the **import** or **export** clause of a **sig** type. Type abbreviations are eliminated from a type or expression with the $|\bullet|_D$ operator, as sketched in Figure 18. The subscript is omitted from $|\bullet|_D$ when D is clear from context.

4.3.2 UNIT$_\mathsf{e}$ Evaluation

Given a type equation of the form **type** $t = \tau$, the variable t can be replaced everywhere with τ once the complete program is known. Since the type system disallows cyclic type definitions, this expansion of types as abbreviations is guaranteed to terminate. Meanwhile, until the complete program is known, type equations are preserved as necessary. In the rewriting semantics for units, type equations are preserved by linking, and then expanded away by invocation. This semantics formalizes the intuition that type equations constrain how programs are linked, but they have no run-time effect when programs are executed.

The reduction rules for UNIT$_\mathsf{e}$ are nearly the same as the rules for UNIT$_\mathsf{d}$ (see Figure 11) or UNIT$_\mathsf{c}$. Like in UNIT$_\mathsf{c}$, UNIT$_\mathsf{e}$'s **invoke** and **compound** reductions propagate **type** definitions as well as **val** definitions. In addition, the **compound** reduction propagates type abbreviations, but the **invoke** reduction immediately expands all type abbreviations in the invoked unit.

5 Other Extensions

Experience with other modules systems, particularly those of ML, suggests further extensions to UNIT$_\mathsf{e}$, such as facilities for exposing the implementation of a type, hiding the type (or parts of the type) of a value, or type sharing. The first two of these extensions are straightforward additions to UNIT$_\mathsf{e}$, but the unit analogue of the last one is less clear. In the following subsections, we briefly discuss each of these concepts.

5.1 Exposing Type Information

The ML module system allows signatures that reveal some information about an exported type [12, 20]. The partially exposed types (or *translucent types*) are used for propagating type dependencies in a way that allows type sharing, but they are also useful for assigning a name to a complex type that is exposed to clients.

Consider exporting values of type *env* from an *Environment* unit such that *env* is revealed as a procedure type. As shown in Figure 20, the translucent type *env* in this case may be viewed as a type abbreviation that is preserved within the signature. The unit *Environment* does not export the type *env*. Instead, the unit and its signature are extended with an extra section that defines the abbreviation *env*. The resulting unit and signature are equivalent to the unit and signature that expands *env* in all type expressions.

5.2 Hiding Type Information

Large projects often have multiple levels of clients. Some of the clients are more trusted than others and are thus privy to more information about the implementation of certain abstractions. To support this situation, UNIT$_\mathsf{e}$ could provide mechanisms for hiding a value's type information from untrusted clients after linking with trusted clients.

$$|\tau|_D = \begin{cases} t & \text{if } \tau{=}t \text{ and } t{\notin}D \\ |\tau'|_D & \text{if } \tau{=}t \text{ and } \langle t = \tau'\rangle \in D \\ |\tau'|_D{\rightarrow}|\tau''|_D & \text{if } \tau{=}\tau'{\rightarrow}\tau'' \\ \textbf{sig import } \overrightarrow{t_i{::}\kappa_i}\ \overrightarrow{x_i{:}|\tau_i|_{D'}} \textbf{ export } \overrightarrow{t_e{::}\kappa_e}\ \overrightarrow{x_e{:}|\tau_e|_{D'}} & \text{if } \tau{=}\mathsf{sig}[i, e, di, de, b] \\ \quad \textbf{depends } \overrightarrow{t_{de} \leadsto t_{di}} & \quad \text{and } D' = \{\langle t = \tau\rangle | \langle t = \tau\rangle \in D \text{ and } t \notin \overrightarrow{t_i}\cup\overrightarrow{t_e}\} \\ \quad |\tau_b|_{D'} & \end{cases}$$

$$|e|_D = \begin{cases} x & \text{if } e{=}x \\ \textbf{unit import } \overrightarrow{t_i{::}\kappa_i}\ \overrightarrow{x_i{:}\tau_i} \quad \textbf{export } \overrightarrow{t_e{::}\kappa_e}\ \overrightarrow{x_e{:}\tau_e} & \text{if } e{=}\textbf{unit import } \overrightarrow{t_i{::}\kappa_i}\ \overrightarrow{x_i{:}\tau_i} \quad \textbf{export } \overrightarrow{t_e{::}\kappa_e}\ \overrightarrow{x_e{:}\tau_e} \\ \quad \overrightarrow{\textbf{type } t_a{::}\kappa_a = |\tau_a|_{D'}} & \quad \overrightarrow{\textbf{type } t_a{::}\kappa_a = \tau_a} \\ \quad \overrightarrow{\textbf{type } t = x_{\mathsf{cl}},x_{\mathsf{dl}}\ |\tau_{\mathsf{l}}|_{D'} \mid x_{\mathsf{cr}},x_{\mathsf{dr}}\ |\tau_{\mathsf{r}}|_{D'} \rhd x_{\mathsf{t}}} & \quad \overrightarrow{\textbf{type } t = x_{\mathsf{cl}},x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}},x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \rhd x_{\mathsf{t}}} \\ \quad \overrightarrow{\textbf{val } x{:}|\tau|_{D'} = |e|_{D'}} \textbf{ in } |e_b|_{D'} & \quad \overrightarrow{\textbf{val } x{:}\tau = e} \textbf{ in } e_b \\ & \text{and } D' = \{\langle t = \tau\rangle | \langle t = \tau\rangle \in D \text{ and } t \notin \overrightarrow{t_i}\cup\overrightarrow{t_e}\cup\overrightarrow{t_a}\cup\overrightarrow{t}\} \\ \ldots & \end{cases}$$

Figure 18: Expanding a type or expression with respect to a set of type abbreviations

$$\frac{\begin{array}{c}\overrightarrow{t_{de}} \subseteq \overrightarrow{t_e} \quad \overrightarrow{t_{di}} \subseteq \overrightarrow{t_i} \quad \Gamma' = \Gamma, \overrightarrow{t_i{::}\kappa_i}, \overrightarrow{t_e{::}\kappa_e} \quad FTV(\tau_b) \cap \overrightarrow{t_e} = \emptyset \\ \Gamma' \vdash \tau_i :: \kappa_i \quad \Gamma' \vdash \tau_e :: \kappa_e \quad \Gamma' \vdash \tau_b :: \Omega\end{array}}{\Gamma \vdash \mathsf{sig}[i, e, di, de, b] :: \Omega}$$

$$\frac{\begin{array}{c}\overrightarrow{t_i}\cup\overrightarrow{t}\cup\overrightarrow{x_i}\cup\overrightarrow{x_{\mathsf{cl}}}\cup\overrightarrow{x_{\mathsf{dl}}}\cup\overrightarrow{x_{\mathsf{cr}}}\cup\overrightarrow{x_{\mathsf{dr}}}\cup\overrightarrow{x_{\mathsf{t}}}\cup\overrightarrow{x} \text{ distinct} \\ \overrightarrow{t_e{::}\kappa_e}\cup\overrightarrow{x_e{:}\tau_e} \subseteq \overrightarrow{t_a{::}\kappa_a}\cup\overrightarrow{t{::}\Omega}\cup\overrightarrow{x{:}\tau}\cup\overrightarrow{x_{\mathsf{cl}}{:}\tau_{\mathsf{l}} \to t}\cup\overrightarrow{x_{\mathsf{dl}}{:}t \to \tau_{\mathsf{l}}}\cup\overrightarrow{x_{\mathsf{cr}}{:}\tau_{\mathsf{r}} \to t}\cup\overrightarrow{x_{\mathsf{dr}}{:}t \to \tau_{\mathsf{r}}}\cup\overrightarrow{x_{\mathsf{t}}{:}t \to \mathsf{bool}} \\ D = \overrightarrow{\langle t_a = \tau_a\rangle} \quad \tau_a \propto_D t'_a \Rightarrow \tau'_a \not\propto_D t_a \text{ for } \langle t_a = \tau_a\rangle, \langle t'_a = \tau'_a\rangle \in D \\ \overrightarrow{t_{de} \leadsto t_{di}} = \{t_a \leadsto t_i \mid \langle t_a = \tau_a\rangle \in D \text{ and } t_i \in \overrightarrow{t_i} \text{ and } t_a \in \overrightarrow{t_e} \text{ and } \tau_a \propto_D t_i\} \\ \Gamma \vdash \mathsf{sig}[i, e, di, de, b] :: \Omega \quad \Gamma' = \Gamma, \overrightarrow{t_i{::}\kappa_i}, \overrightarrow{t{::}\Omega} \quad \Gamma'_a = \Gamma', \overrightarrow{t_a{::}\kappa_a} \quad \Gamma'_a \vdash \tau_a :: \kappa_a \\ \Gamma' \vdash |\overrightarrow{\tau_{\mathsf{l}}}| :: \Omega \quad \Gamma' \vdash |\overrightarrow{\tau_{\mathsf{r}}}| :: \Omega \quad \Gamma' \vdash |\overrightarrow{\tau}| :: \Omega \\ \Gamma'' = \Gamma', \overrightarrow{x_i{:}|\tau_i|}, \overrightarrow{x_{\mathsf{cl}}{:}|\tau_{\mathsf{l}}| \to |t|}, \overrightarrow{x_{\mathsf{dl}}{:}|t| \to |\tau_{\mathsf{l}}|}, \overrightarrow{x_{\mathsf{cr}}{:}|\tau_{\mathsf{r}}| \to |t|}, \overrightarrow{x_{\mathsf{dr}}{:}|t| \to |\tau_{\mathsf{r}}|}, \overrightarrow{x_{\mathsf{t}}{:}|t| \to \mathsf{bool}} \quad \Gamma'' \vdash_{\mathsf{s}} |\overrightarrow{e}| : |\overrightarrow{\tau}| \quad \Gamma'' \vdash |e_b| : \tau_b\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{unit import } \overrightarrow{t_i{::}\kappa_i}\ \overrightarrow{x_i{:}\tau_i} \quad \textbf{export } \overrightarrow{t_e{::}\kappa_e}\ \overrightarrow{x_e{:}\tau_e} \\ \qquad \overrightarrow{\textbf{type } t_a{::}\kappa_a = \tau_a} \\ \qquad \overrightarrow{\textbf{type } t = x_{\mathsf{cl}},x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}},x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \rhd x_{\mathsf{t}}} \\ \qquad \overrightarrow{\textbf{val } x{:}\tau = e} \textbf{ in } e_b \\ \quad : \mathsf{sig}[i, e, di, de, b]\end{array}}$$

$$\frac{\begin{array}{c}\overrightarrow{t_i}\cup\overrightarrow{t_{p1}}\cup\overrightarrow{t_{p2}}\cup\overrightarrow{x_i}\cup\overrightarrow{x_{p1}}\cup\overrightarrow{x_{p2}} \text{ distinct} \quad \overrightarrow{t_e}\cup\overrightarrow{x_e} \text{ distinct} \\ \overrightarrow{t_{w1}{::}\kappa_{w1}}\cup\overrightarrow{x_{w1}{:}\tau_{w1}} \subseteq \overrightarrow{t_i{::}\kappa_i}\cup\overrightarrow{t_{p2}{::}\kappa_{p2}}\cup\overrightarrow{x_i{:}\tau_i}\cup\overrightarrow{x_{p2}{:}\tau_{p2}} \\ \overrightarrow{t_{w2}{::}\kappa_{w2}}\cup\overrightarrow{x_{w2}{:}\tau_{w2}} \subseteq \overrightarrow{t_i{::}\kappa_i}\cup\overrightarrow{t_{p1}{::}\kappa_{p1}}\cup\overrightarrow{x_i{:}\tau_i}\cup\overrightarrow{x_{p1}{:}\tau_{p1}} \\ \overrightarrow{t_e{::}\kappa_e}\cup\overrightarrow{x_e{:}\tau_e} \subseteq \overrightarrow{t_{p1}{::}\kappa_{p1}}\cup\overrightarrow{t_{p2}{::}\kappa_{p2}}\cup\overrightarrow{x_{p1}{:}\tau_{p1}}\cup\overrightarrow{x_{p2}{:}\tau_{p2}} \\ \Gamma \vdash e_1 : \mathsf{sig}[i1, e1, di1, de1, b1] \quad \Gamma \vdash e_2 : \mathsf{sig}[i2, e2, di2, de2, b2] \\ \Gamma \vdash \mathsf{sig}[w1, p1, di1, de1, b1] :: \Omega \quad \Gamma \vdash \mathsf{sig}[w2, p2, di2, de2, b2] :: \Omega \\ \mathsf{sig}[i1, e1, di1, de1, b1] \leq \mathsf{sig}[w1, p1, di1, de1, b1] \quad \mathsf{sig}[i2, e2, di2, de2, b2] \leq \mathsf{sig}[w2, p2, di2, de2, b2] \\ \Gamma \vdash \mathsf{sig}[i, e, di, de, b2] :: \Omega \quad \overrightarrow{\langle t_{di1}, t_{de1}\rangle} \cap \overrightarrow{\langle t_{de2}, t_{di2}\rangle} = \emptyset \\ \overrightarrow{t_{de} \leadsto t_{di}} = \{t_e \leadsto t_i \mid t_i \in \overrightarrow{t_i} \text{ and } t_e \in \overrightarrow{t_e} \text{ and } t_e \leadsto t_i \in \overrightarrow{t_{de1} \leadsto t_{di1}}\cup\overrightarrow{t_{de2} \leadsto t_{di2}}\}\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{compound import } \overrightarrow{t_i{::}\kappa_i}\ \overrightarrow{x_i{:}\tau_i} \textbf{ export } \overrightarrow{t_e{::}\kappa_e}\ \overrightarrow{x_e{:}\tau_e} \\ \qquad \textbf{link } e_1 \textbf{ with } \overrightarrow{t_{w1}{::}\kappa_{w1}}\ \overrightarrow{x_{w1}{:}\tau_{w1}} \textbf{ provides } \overrightarrow{t_{p1}{::}\kappa_{p1}}\ \overrightarrow{x_{p1}{:}\tau_{p1}} \\ \qquad \textbf{and } e_2 \textbf{ with } \overrightarrow{t_{w2}{::}\kappa_{w2}}\ \overrightarrow{x_{w2}{:}\tau_{w2}} \textbf{ provides } \overrightarrow{t_{p2}{::}\kappa_{p2}}\ \overrightarrow{x_{p2}{:}\tau_{p2}} \\ \quad : \mathsf{sig}[i, e, di, de, b2]\end{array}}$$

Figure 19: Type checking for $\textsc{Unit}_{\mathsf{e}}$

Consider the example in Figure 21. The *Environment* unit is linked with the *Letrec* unit, allowing the latter to exploit the implementation of environments as procedures. In contrast, other clients should not be allowed to exploit the implementation of environments. Hence, the type of environments should be opaque outside the compound unit *RecEnv*, which combines *Environment* and *Letrec*.

As shown in Figure 21, information about *RecEnv*'s exports can be restricted via explicit signatures and an extended subtype relation. The extended relation allows a subtype signature to contain an extra exported type variable (*e.g.*, *env*) in place of an abbreviation in the supertype

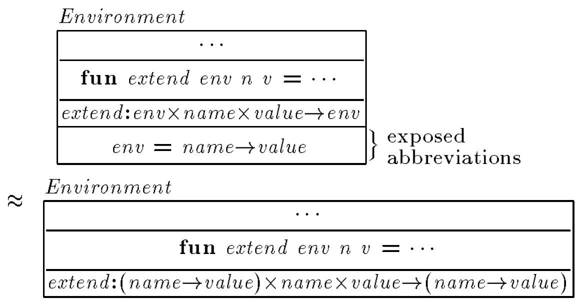

Figure 20: Exposing information for a type

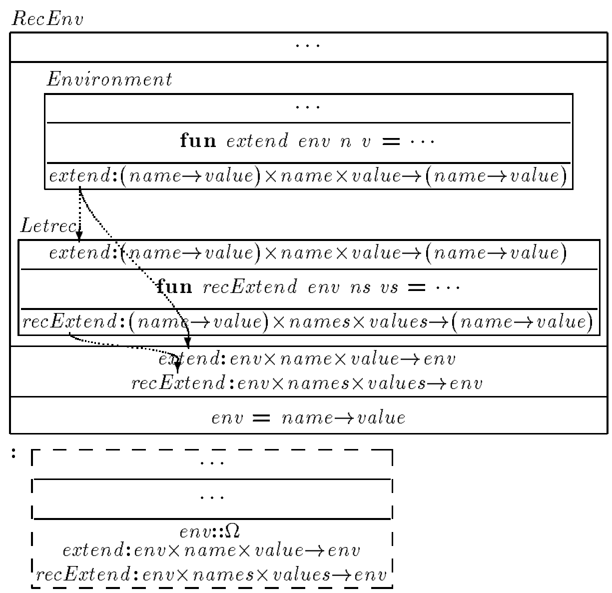

Figure 21: Hiding type information for an exported value

signature. As a result, the information formerly exposed by the abbreviation becomes hidden, replaced by an opaque type.

5.3 Sharing

Type sharing specifications are used to solve the "diamond import" problem for ML [22]. Suppose a particular *symbol* structure is provided to both a *lexer* functor and a *parser* functor. If both *lexer* and *parser* export the type sym as originating from *symbol* (via translucent types), the structures returned by *parser* and *lexer* can be joined by a functor that accepts structure arguments agreeing on the sym type. This "agreement" prerequisite is declared via a type sharing specification.

In UNIT$_e$, the "diamond import" problem is solved by linking *lexer*, *parser*, and *symbol* together at once. However, the unit model provides nothing like after-the-fact sharing specifications; thus, if *lexer* and *parser* are compound units that contain internal instances of *symbol*, then *symbol* is instantiated twice and there is no way to unify the two sym types.

Type sharing is more flexible than unit linking in one important case. Suppose that *lexer* links to many structures and has many clients. In addition, suppose that most of the clients share types with only a few of the structures. In ML, each client can conveniently declare a few after-the-fact sharing declarations. In UNIT$_e$, if *lexer* requires many imports, each of *lexer*'s clients must provide all of those imports, regardless of how few imports need to be shared. This example illustrates a problem of specifying *which* units need to be linked, rather than *how* to link them. As explained in the introduction, integrating units with the core language provides power for selecting *which* units to link. Future work must explore how core language features can be used to express complex linking patterns for units, and whether our model needs adjustment to accommodate common re-use patterns.

6 Related Work

As already mentioned in Section 2, our unit model incorporates ideas from distinct language communities, particularly those using packages and ML-style modules. The Scheme and ML communities have produced a large body of work exploring variations on the standard module system, especially variations for higher-order modules [2, 4, 12, 16, 18, 20, 21, 23, 29]. Duggan and Sourelis [5] have investigated "mixin modules" for specifying recursive and extensible definitions across modules; their work and ours have no overlap.

Cardelli [3] anticipated the unit language's emphasis on module linking as well as module definition. Our unit model is more concrete than his proposal and addresses many of his suggestions for future work. Kelsey's proposed module system for Scheme [17] captures most of the organizational properties of units, but does not address static typing or dynamic linking.

7 Conclusion

Program units deliver both the traditional benefits of modules for separate compilation and the more recent advances of higher-order modules and programmer-controlled linking. Our unit model also addresses the often overlooked, but increasingly important, problem of dynamic linking.

The unit language was originally implemented for the development of DrScheme [7, 25], Rice's Scheme programming environment, which is implemented using MzScheme. DrScheme is a large and dynamic program with many integrated components, including a multimedia editor, an interactive evaluator, a syntax checker, and a static debugger. Additional components can be dynamically linked into the environment. DrScheme also acts as an operating system for client programs that are being developed, launching client programs by dynamically linking them into the system while maintaining the boundaries between clients.

Future work must focus on making units syntactically practical for typed languages. Our text-based model is far too verbose, and we do not address the design of a linking language. Instead, we provide a simple construct for linking units and rely on integration with the core language to build up linking expressions. This integration simplifies our presentation, and we believe it is an essential feature of units. Nevertheless, future work must explore more carefully the implications of integrating the core and module languages.

Acknowledgements The authors would like to thank Robby Findler for early contributions to this work, and Cormac

Flanagan, Bob Harper, Richard Kelsey, Shriram Krishnamurthi, Peter Lee, Didier Rémy, Scott Smith, Paul Steckler, and Jérôme Vouillon for stimulating discussion and comments on the paper. Thanks also to the anonymous reviewers for their comments. Special thanks to Shriram for suggesting the title.

References

[1] Barnes, J. G. P. *Programming in Ada 95.* Addison-Wesley, 1996.

[2] Biswas, S. K. Higher-order functors with transparent signatures. In *Proc. ACM Symposium on Principles of Programming Languages* (1995), pp. 154–163.

[3] Cardelli, L. Program fragments, linking, and modularization. In *Proc. ACM Symposium on Principles of Programming Languages* (1997), pp. 266–277.

[4] Curtis, P., and Rauen, J. A module system for Scheme. In *Proc. ACM Conference on Lisp and Functional Programming* (1990), pp. 13–28.

[5] Duggan, D., and Sourelis, C. Mixin modules. In *Proc. ACM International Conference on Functional Programming* (1996), pp. 262–273.

[6] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. Tech. Rep. 100, Rice University, June 1989. *Theoretical Computer Science*, volume 102, 1992, pp. 235–271.

[7] Findler, R. B., Flanagan, C., Flatt, M., Krishnamurthi, S., and Felleisen, M. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs* (1997), pp. 369–388.

[8] Flatt, M. PLT MzScheme: Language manual. Tech. Rep. TR97-280, Rice University, 1997.

[9] Flatt, M., Krishnamurthi, S., and Felleisen, M. Classes and mixins. In *Proc. ACM Symposium on Principles of Programming Languages* (1998), pp. 171–183.

[10] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[11] Harbison, S. P. *Modula-3.* Prentice Hall, 1991.

[12] Harper, R., and Lillibridge, M. A type-theoretic approach to higher-order modules with sharing. In *Proc. ACM Symposium on Principles of Programming Languages* (1994), pp. 123–137.

[13] Harper, R., Mitchell, J., and Moggi, E. Higher-order modules and the phase distinction. In *Proc. ACM Symposium on Principles of Programming Languages* (1990), pp. 341–354.

[14] Harper, R., and Stone, C. A type-theoretic semantics for Standard ML 1996. Submitted for publication, 1997.

[15] Hudak, P., and Wadler, P. (Eds.). Report on the programming language Haskell. Tech. Rep. YALE/DCS/RR777, Yale University, Department of Computer Science, Aug. 1991.

[16] Jagannathan, S. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 456–492.

[17] Kelsey, R. A. Fully-parameterized modules or the missing link. Tech. Rep. 97-3, NEC Research Institute, 1997.

[18] Lee, Shinn-Der and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proc. ACM Symposium on Principles of Programming Languages* (1993), pp. 479–492.

[19] Leroy, X. Unboxed objects and polymorphic typing. In *Proc. ACM Symposium on Principles of Programming Languages* (1992), pp. 177–188.

[20] Leroy, X. Manifest types, modules, and separate compilation. In *Proc. ACM Symposium on Principles of Programming Languages* (1994), pp. 109–122.

[21] Leroy, X. Applicative functions and fully transparent higher-order modules. In *Proc. ACM Symposium on Principles of Programming Languages* (1995), pp. 142–153.

[22] MacQueen, D. Modules for Standard ML. In *Proc. ACM Conference on Lisp and Functional Programming* (1984), pp. 198–207.

[23] MacQueen, D. B., and Tofte, M. A semantics for higher-order functors. In *European Symposium on Programming* (Apr. 1994), Springer-Verlag, LNCS 788, pp. 409–423.

[24] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

[25] Rice University PLT. *DrScheme.* URL: `www.cs.rice.edu/CS/PLT/packages/drscheme/`.

[26] Saraswat, V. Java is not type-safe, Aug. 1997. URL: `www.research.att.com/~vj/bug.html`.

[27] SunSoft. *SunOS 5.5 Linker and Libraries Manual*, 1996.

[28] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM Conference on Programming Language Design and Implementation* (1996), pp. 181–192.

[29] Tofte, M. Principal signatures for higher-order program modules. In *Proc. ACM Symposium on Principles of Programming Languages* (1992), pp. 189–199.

[30] Wirth, N. *Programming in Modula-2.* Springer-Verlag, 1983.

[31] Wright, A., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 160, Rice University, 1991. *Information and Computation*, volume 115(1), 1994, pp. 38–94.