Pomona College
Department of Computer Science

# Information Hiding and Expressiveness in *LOOM* Modules

Nathan Reed

May 1, 2008

Submitted as part of the senior exercise for the degree of
Bachelor of Arts in Computer Science

Professor Kim Bruce, advisor

## Abstract

We examine the problems associated with incorporating useful and expressive modularity into statically typed, object-oriented programming languages. We propose a set of general criteria against which module systems in such languages may be judged, and discuss where existing languages (including $\mathcal{LOOM}$) fall short of meeting these criteria.

We then extend the module system of the $\mathcal{LOOM}$ language to improve its expressiveness while maintaining strong information hiding and separate compilability, by allowing each module to reveal multiple interfaces to the outside world. There is no restriction on these interfaces beyond that they be consistent with one another and with definitions in the module implementation.

Finally, we describe how our type-checker for the extended language has been implemented, using a constraint graph data structure to represent information about user-defined types that cross module boundaries.

# Contents

# List of Figures

# Chapter 1

# Introduction

A *modular program* is one that is split into pieces, called *modules*, which each contain a subset of the program's functionality. Modules work together to form a complete program, but at the same time each module is a self-contained entity. A module groups together some set of related definitions, such as types, classes, functions or procedures, and global variables; the module then provides an abstraction of the functionality it contains by exposing an interface that clients (other modules, or top-level application code) can access.

Since modularity has been recognized as an important principle of software design [Par72, McC04], many programming languages have included built-in support for it. In §2, we discuss the benefits of modularity and then propose a set of criteria that well-designed module systems should satisfy. We emphasize lexical separation of interfaces and implementations, strong information hiding, fine-grained namespace management, and support for Meyer's open/closed principle [Mey97]. Some languages, such as Java and C++, have a rich concept of classes that combines traditional object-oriented features with additional features to support modularity. However, we argue that classes and modules should remain separate, with [Szy92]. We then review the module systems of several important languages and discuss where they fall short of our criteria.

In the following part of this work, we focus on a language called $\mathcal{LOOM}$ that has been developed by Professor Kim Bruce and various students [Bru96, BFP97, BPV98]. $\mathcal{LOOM}$ is a statically typed, provably type-safe, object-oriented language with a basic module system. In §3 we summarize the most relevant features of $\mathcal{LOOM}$ and then discuss the shortcomings of its original module system. Although $\mathcal{LOOM}$'s original module system was

statically type-safe and provided good support for separate compilation, interface/implementation separation, and strong information hiding, it was insufficiently expressive to support the open/closed principle.

In §4, we describe an extension to $\mathcal{LOOM}$ that greatly improves the expressiveness of modules while maintaining separate compilability and as much information hiding as possible. This is done by allowing each module to expose multiple independent interfaces to the outside world. A module can provide, for instance, a minimal interface for ordinary users and a more detailed interface for clients who wish to extend the module; a module that faces several clients in a large software system can also provide a distinct interface for each client, specialized for that client's needs.

The module system we propose is sufficiently general that implementing a type-checker for it is not completely straightforward. In particular, $\mathcal{LOOM}$ interfaces are allowed to specify fairly general constraints on user-defined types in order to reveal the minimum of information that clients need, without "leaking" any implementation details into the interface. When a module or program imports a set of interfaces, the constraints they declare must be checked for consistency, which is nontrivial. Therefore, in §4.3 we describe in some detail our algorithm for processing a graph of the constraints among a set of imported types, which checks the graph for consistency and also regularizes its structure in such a way as to make the graph simpler and more efficient for the type-checker to use.

We will conclude by arguing that our extension of $\mathcal{LOOM}$ addresses one of the major shortcomings of the original module system, and then suggesting future directions of work in this area.

# Chapter 2

# Background

## 2.1 Benefits of modularity

Modularity is an important design principle in software engineering because a well-modularized program has greater *comprehensibility*—it is easier for humans to understand [McC04]. A large, complicated program likely has far too many details for a single person to remember. Modules provide large-scale internal structure to a program by breaking it into more conveniently sized chunks that are small enough for individuals to understand, and sufficiently decoupled from the rest of the program that they can be understood in isolation. Moreover, modularity can make it possible for a person to understand how the program functions as a whole by understanding the modules' interfaces and how the modules interact with one another, without needing to know how each module works internally. Modularity enables developers on a project to work together more effectively, since each person can become an expert on just a few of the program's modules, without needing to understand the details of all the others. Because well-modularized programs are easier to understand, they are also easier to construct, and tend to contain fewer bugs.

In addition to its benefits for comprehensibility, modular programming also improves code *reusability*. A well-designed module has few dependencies on specific details of the program in which it was originally developed, and those dependencies it has are clearly stated; this lets it be more easily re-used in another program.

Modules also benefit *maintainability*, because separating the implementation of a module from its interface means that changes to the module that do not affect its interface can be made without affecting any other modules

in the program. Modules limit the scope of the unfortunate "domino effect" wherein changing one aspect of a program breaks several others, necessitating fixes that in turn break more features and so forth, causing a wave of change that propagates throughout the program. Likewise, since modules interact only through their interfaces, modularity makes it easier to construct different editions or variants of a program, by simply plugging in different implementations for the same interface. For instance, modularity can be used to improve portability, if the platform-specific code in a program is isolated in a module that can be rewritten for each new platform without changing the rest of the program.

Also, modules enable *encapsulation* and *information hiding* by providing natural boundaries along which these policies can be enforced. Encapsulation and information hiding improve program design by preventing different pieces of the program from taking advantage of each other's implementation details [Par72]. Since modules already separate programs into pieces with interfaces and implementations, it makes sense for programming languages with module systems to enforce encapsulation and information hiding by making it illegal for code to access the routines or state hidden in another module's implementation.

Finally, another benefit of modular programming is *separate compilation*. Partitioning a program into modules should enable each module to be compiled independently of the others. This allows a programmer to make a change to a program and then recompile only those modules affected, which can vastly improve compilation times (and therefore programmer productivity) in large projects. Separate compilation can also make modules natural units for dynamic linking [Szy92].

## 2.2 Features of a good module system

Since modularity is so beneficial for software engineering, programming language developers must consider how they can design languages with good support for modular programming. There are some features that any well-designed module system should have.

First, a module system should allow for clean lexical separation between interfaces and implementations. The whole point of a module system is that the connections between different modules are clearly stated, and a programmer need not read through the implementation of a module to discover what its interface contains. Moreover, in the real world, source code is not always the best format in which to distribute software; interfaces and

implementations must be able to live in separate source files in order to support binary distribution of libraries.

A good module system should support information hiding. That is, it must be expressive enough that the interface can contain only the information needed for clients of the module to use it; no details of the implementation should "leak" into the interface. One way of doing this is to allow interfaces to define *abstract data types* (ADTs). With an ADT, a type is declared in an interface but no information other than its name is given, making it an *opaque type*. To allow clients to use the type without knowledge of its internal structure, the interface also reveals functions that create and operate upon values of the opaque type. However, in object-oriented languages, ADT-style information hiding is clunky; we would prefer to allow clients to operate by calling methods of an object rather than passing opaque values to free functions. One way to achieve this is to support *partial revelation* [Pet96, Fre95], in which the name and some of the methods of an object type are given in an interface, while the full definition is hidden in the implementation. We will say more about partial revelation in §3.2.2.

Despite the demands of information hiding, the module system should not break static typing, and it should enable separate compilation. Ideally, a module can be type-checked and compiled entirely in isolation, knowing nothing but the interfaces of any other modules it imports. This implies that modules can be compiled in any order and that the correctness of a module's compilation cannot depend on knowing implementation details of other modules. However, it also implies that an interface must contain enough information to type-check and compile modules that import it.

Another concern related to modules is *namespace pollution* [Pet96]. Importing a module usually introduces the names defined by the module's interface into the importing scope. However, in a piece of code that imports many modules, it may become difficult to keep track of which names came from where, particularly if the imported modules have large interfaces. It is also possible for name collisions to occur, e.g. when two modules define symbols with the same name. To sidestep these problems, a module system can provide different modes of import. For instance, there could be one mode in which imported declarations must be referred to by their fully-qualified names (names that include the name of the module they came from, as a prefix) and another mode in which the programmer explicitly states which names should be usable in short (not fully-qualified) form.

Finally, a good module system should support the *open/closed principle*. Meyer originally stated this principle for classes [Mey97], but we here extend it to cover modules: *modules should be open to extension, but closed to*

*modification*. What this means is that once a well-designed module has been constructed, it should not be necessary to modify it to extend it with new features; rather, new features should be implemented by creating new modules that can be imported alongside the old module. This has some interesting implications for object-oriented languages. In a good information-hiding module design, object types will frequently be revealed only partially in the interface (see §3.2.2). This partial revelation gives clients the information they need to *use* the object type; however, to *extend* an object type we will generally need a full revelation of all its fields and methods. Therefore, supporting the open/closed principle requires languages to either provide a way to bypass a module's information hiding, or support multiple interfaces for a module (e.g. one for normal clients and one for extenders). Each alternative has its advantages and disadvantages. We will have more to say about this in §3.3.

## 2.3 Why classes don't make good modules

Classes and modules have a good deal in common. They both act as containers for sets of related definitions, they both introduce a new namespace for the names they define, and they both create an abstraction of the functionality they contain. Several object-oriented languages, such as C++, Java, Eiffel, and Smalltalk [Szy92], have therefore attempted to use classes for modularization by giving classes the ability to hide information and in some cases making them the units of separate compilation. However, there are good reasons not to conflate classes and modules.

The primary purpose of classes is to generate objects. Methods of a class do not work like ordinary functions; they accept the object on which to act as an implicit parameter. However, there are many kinds of functions that it makes sense to group together, but for which the idea of acting on an object makes little sense. An example is mathematical functions [Szy92]. If functions like `sin`, `cos`, `sqrt` and so forth are packaged into a class, we will have to instantiate an object of the class in order to call these functions. But the functions do not depend on the object and have no need of the object, making the instantiation superfluous. To avoid this problem, Java and C++ have introduced "static" methods, which can be called without an object [Str00, AGH05]. In Eiffel, which does not provide static methods, the convention is to inherit the class in which desired functions are defined; for instance, a class needing to access `sqrt` would inherit the `Math` class (of which `sqrt` is a method). Eiffel provides features to avoid name conflicts and

namespace pollution, such as the ability to selectively rename or hide inherited definitions [Eif06]. However, both static methods and renaming/hiding inherited definitions are band-aids for the fundamental problem, which is that classes are not an appropriate construct to modularize functionality that does not fit into the object model.

Moreover, using classes as modules can lead to the problem of "spaghetti scoping", wherein a class hides its private information from most other classes in the system, but exposes it to a specified set of other classes. This situation arises when some group of classes need to maintain a group invariant [Szy92]. The classes must hide information to prevent client code from directly manipulating their internal representations and possibly breaking the invariant; however, they must reveal this information to each other to maintain the invariant. In C++ this can be accomplished by making the classes "friends" of each other. Another possibility is to allow nested classes, as Java, C++, and D do. However, class nesting is somewhat anomalous, in that it is unclear how objects of inner classes should be instantiated. In C++, inner classes are simply classes that are only visible in the namespace of an outer class, while in Java and D, inner class instances can be associated with a specific instance of the outer class and can access that outer object's instance variables and private methods [Str00, AGH05, Bri07]. Both friends and nested classes are less natural than simply putting related classes in a module, which would allow the classes to expose implementation details to one another but hide them from clients of the module.

Although concepts like abstraction and information hiding make sense for both classes and modules, the two are sufficiently different that trying to unify them leads to an unnecessarily complicated class system and a poor module system. Therefore, modules in object-oriented languages are better implemented as a separate language feature orthogonal to classes, not by overloading the class concept to enable modularity.

## 2.4 Existing module systems

In this section, we will briefly summarize the modularity features of some relevant languages and point out their shortcomings.

### 2.4.1 C++

C++ has no module system to speak of. C++ programs are modularized by convention, i.e. by having interface declarations in header files and implementations in source files. Importing is done with the preprocessor, which

simply copies the contents of header files verbatim into source files. As a consequence, earlier included declarations can affect the semantics of later declarations, causing headaches wherein includes must occur in a specific order for the program to compile. Moreover, limitations of the language mean that not all implementation details can be hidden with headers; in particular, class definitions in header files must include all instance variables (and therefore, all the definitions of the types of instance variables, etc.) so that client code can determine how much memory to allocate for an instance of the class.[1] C++ does include `namespace` and `using` declarations, which allow programmers to create new namespaces on the fly and import names from one namespace into another. While useful for preventing namespace pollution, this feature does not mitigate the previously mentioned problems. Ultimately, modularization in C++ is *ad hoc*, not built into the language in a coherent way.

### 2.4.2 Java

Java, like C++, gives classes many of the features of modules. In addition, Java makes classes the units of separate compilation and dynamic linking, and provides packages, which group together related classes or sub-packages. However, Java packages are defined simply by including a `package` statement in each source file, which makes it possible for any client to add their own code to a package, obtaining access to the package's internal definitions. Moreover, interfaces for packages are not separately declared; programmers must read a package's source code to determine what is in its interface (or use an automated tool to do it for them). Java does not require explicit importing either, since fully-qualified names can always be used to refer to any class whose definition can be found by searching the directories listed in the `CLASSPATH` environment variable. Finally, though classes can usually be separately compiled, mutually recursive classes cannot; they must be compiled at the same time, which adds complexity and confusion to the compilation process.

Lujo Bauer, Andrew Appel, and Edward Felten have worked on an extension to Java that improves its module system [BAF03]. Their system fixes several of the problems discussed above: modules contain explicit lists of their source files, the modules they import, and the classes they export. Requiring a list of sources fixes the problem in which any client can add their own code to a package; explicitly listing imported modules is a step

[1] The C++ community has developed ways to work around this particular issue, such as the "`pImpl` idiom" and the factory pattern.

in the right direction in terms of making sure a programmer can tell where an imported symbol is coming from. However, this system still does not require fully explicit module interfaces; only the names of exported classes are given in the module description, but not their fields or methods.

Another interesting feature of Bauer, Appel, and Felten's work is their use of hierarchical modularity, which allows a single module to be given multiple interfaces by wrapping it in other modules that re-export selected parts of it. It is not clear, however, whether clients can recognize the same type when it is available under different two or more names simultaneously (aliasing). We will have more to say about aliasing in §3.4.

### 2.4.3 Python

Python is a dynamic programming language that has gained a good deal of popularity in recent years. It has a module system similar to Java's packages, but with no information hiding at all. There is a distinction between public and non-public definitions in a module, but it is only used to determine which names to introduce into the importing code's namespace. Non-public definitions can still be accessed from outside a module using fully-qualified names. Like Java, it also has the problem that module interfaces are not declared separately from the module's source code. However, explicit import declarations are required to make a module's definitions available even with fully-qualified names. Another mode of import allows selected definitions to additionally be accessed by abbreviated names.

### 2.4.4 D

D is a relatively new language intended to provide an alternative to C and C++. It is designed to be suitable for high-performance applications and low-level systems programming, but also attempts to incorporate some of the high-productivity features from modern scripting languages [Bri07]. D modules are quite similar to Java packages and Python modules, but D allows greater control over the import process, providing several different ways to import a module with slightly different effects. D modules are separately compilable, require explicit import, and support information hiding. However, they again do not lexically separate module interfaces and implementations, and D classes exposed in a module interface are subject to the same limitation as in C++, where all the instance variables must be known to clients so that the right amount of space can be allocated for new objects.

### 2.4.5 Modula-3

Modula-3 is an extension of Modula-2 that adds support for object-oriented programming (as well as other new features). It has a quite strong module system, a feature inherited from its predecessors, with good support for interface/implementation separation and information hiding. Like Python, it provides two forms of import, in which one imports only fully-qualified names while the other allows abbreviated names.

Modula-3 also allows a module to have multiple interfaces, which allow modules to reveal different information to different clients. This supports the open/closed principle. However, the Modula-3 system is still not as general as it could be, since interfaces must be nonoverlapping [Nel91]. Moreover, Modula-3's requirement that keywords be all uppercase makes it, for many people, a very annoying language.

### 2.4.6 Standard ML

Standard ML (SML), though it is not object-oriented, has a quite general and flexible module system. Interfaces, which are called *signatures*, are well-separated from implementations, called *structures*. A signature can be *ascribed* to a structure, hiding all declarations in the structure that are not listed in the signature. Multiple signatures can be ascribed to a structure, and `sharing` declarations can be used to ensure that multiple revelations of the same underlying type can be used interchangeably. SML also has parameterized modules, called *functors*; their imports are not fully resolved until they are instantiated to produce a structure. Nested structures are also supported.

Unfortunately, SML does not properly support separate compilation.[2] It also does not allow fine-grained control of imports. Fully-qualified names defined by other structures in the compilation group are always available without an explicit import statement, and abbreviated names can only be used after an `open` statement, which indiscriminately introduces all of the names in a structure's signature into the importing scope.

### 2.4.7 Moby

Moby [FR02, FR03] is an experimental language designed by Kathleen Fisher and John Reppy. Moby is based on ML, but adds object-oriented features. As such, it inherits both the advantages and disadvantages of ML's

[2] Though SML of New Jersey's built-in `CM` tool provides some incremental compilation.

module system. MOBY's classes support public/private annotations on their members, in addition to partial revelation of object types and class types in signatures. This gives MOBY programmers a good deal of control over visibility and accessibility, as class members can be either hidden entirely from the outside of a module, revealed as 'private' in the interface (visible to everyone, but accessible to extenders only), or revealed as 'public' in the interface (visible and accessible to everyone). On the other hand, this richness provides perhaps slightly more flexibility than needed, and has the potential to make MOBY more confusing to a programmer than a less flexible system might be. Moreover, MOBY allows certain anomalies, such as methods that are visible in a base class but hidden in a subclass; this makes MOBY's information-hiding system incompatible with the `MyType` construct used in $\mathcal{LOOM}$ (see §3.1.2).

### 2.4.8 Units and Mixins

Robert Findler and Matthew Flatt describe an intriguing system that introduces *units* (modules) and *mixins* (classes) in the context of MzScheme, an object-oriented Scheme variant [FF99]. Findler and Flatt's principle is that modules should not specify the source of the definitions they import. In other words, units are effectively modules parameterized by their imports; interfaces that the imports must provide are specified, but multiple implementations of those imports can coexist in the same program. The language includes *linking expressions* that wire one unit's exports to another unit's imports. This is something like SML's functors, but an important difference is that a unit does not have to be recompiled when its imports are changed. So, client code that imports a module can be re-used with an extended version of that module by altering the linking expression, without altering any client code.

Parameterized imports have implications for classes; when a class inherits from a base class defined in an imported module, it is effectively parameterized by its base class, making it what Findler and Flatt term a mixin. This makes units and mixins an elegant way to solve the expression problem [Wad98]. Given a unit that defines the base class for the variants and the initial set of variants and operations, new variants can be added in the usual way by providing new units that define additional subclasses, while new operations can be added in the usual way by subclassing the existing variants. However, unlike in most other languages, we can now go back and combine the new operations with the new variants by making use of the mixin feature to replace the superclass of the new variants with the

extended superclass that defines the new operations. (Of course, we do need to write more code to actually implement the new operations for the new variants.) Then client code that depends on the added operations can make use of added variants without recompiling them, even if the authors of the added variants did not anticipate this.

Units and mixins are elegant, powerful constructs, but since MzScheme is a dynamically typed language, Findler and Flatt did not have to worry too much about breaking static typing. If this sort of functionality is to be introduced into $\mathcal{LOOM}$, its interaction with static typing will have to be very carefully considered.

# Chapter 3

# $\mathcal{LOOM}$ and its modules

In the following sections we'll review the original state of the $\mathcal{LOOM}$ language, and identify where its module system falls short of meeting the criteria we outlined in §2.2. We will then discuss possible approaches to addressing the biggest problem with $\mathcal{LOOM}$ modules, and give examples to clarify some of the issues involved.

## 3.1 Core language

$\mathcal{LOOM}$ is a statically typed, provably type-safe, object-oriented language designed by Kim Bruce and others. $\mathcal{LOOM}$ has been described in detail elsewhere [Bru96, Pet96, BFP97, Van97, BPV98, Thu02]. In this section we'll review its most important aspects.

### 3.1.1 Object types and class types

In $\mathcal{LOOM}$, object types and class types are distinct. An object type is something like an `interface` (note: not a module interface) in Java or D, or an abstract class in C++: it contains a set of (virtual) method signatures, but no fields and no implementation. Classes, which contain fields and implementations, are first-class values in $\mathcal{LOOM}$, and class types are the types of these values; they include the signatures of all methods and the types of all fields. Each class type has a corresponding object type generated by removing all fields and hidden methods from the class type. The following listing shows an example of a class and its corresponding object type:

**Listing 3.1** (Class types and object types)**.**

```
-- A class
class ListNodeClass
  var
    value = 0    : integer;
    next  = nil  : mytype;
  methods
    procedure setValue (v : integer)
      begin
        value := v;
      end;
    function getValue () : integer
      begin
        return value;
      end;
    procedure setNext (n : mytype)
      begin
        next := n;
      end;
    function getNext () : mytype
      begin
        return next;
      end;
end;

-- The type of objects generated by the preceding class
ListNode = objecttype
  setValue  : proc (integer);
  getValue  : func (): integer;
  setNext   : proc (mytype);
  getNext   : func (): mytype;
end;
```

Multiple classes can generate the same object type, and objects of these classes will be interchangeable. Object types have reference semantics, so if $A$ is an object type then variables of type $A$ store either a reference to an object of type $A$, or the special value `nil`, indicating a null reference.

### 3.1.2 `MyType`

*LOOM*'s type system includes a construct called `MyType`, which can be used to define recursive object types. Within an object type, `MyType` stands for the object type itself, i.e. for the type of the implicit `self` variable available inside method bodies.[1]

When defining a class, `MyType` stands for the object type generated by the class. However, when a class is extended, the meaning of `MyType` changes automatically in the subclass. Consider Listing 3.2:

**Listing 3.2** (`MyType` and inheritance)**.**

```
-- A superclass
class Foo
  var ...
  methods
    function equals (other : mytype) : bool
      begin ... end;
    procedure clone () : mytype
      begin ... end;
end;

-- A subclass
class Bar inherits Foo
  methods
    function someNewMethod ...
end;
```

In class `Foo`, the `equals` method takes an object of the same object type that `Foo` generates, and the `clone` method returns an object of this type. In `Bar`, a new visible method is defined, so the object type generated by `Bar` is not the same as that of `Foo`. The inherited `equals` will now only accept objects of the object type generated by `Bar`, and `clone` will now return objects of this type, even though neither method has been redefined in the subclass.

As the example demonstrates, `MyType` is particularly useful for expressing the type of so-called *binary methods*, which are methods that take a parameter of the same type as the receiver. Binary methods occur frequently, but are difficult to type in most languages, because the covariant

[1] This isn't quite true; actually, the type of `self` allows access to instance variables, while `MyType` doesn't.

specialization of the parameter type breaks subtyping [BCC+95]. In the following section we will see how $\mathcal{LOOM}$ works around this problem.

### 3.1.3 Matching

A type $B$ is a *subtype* of type $A$, written $B <: A$, if every element of $B$ can "masquerade" as an element of $A$; that is, if any code that works with an $A$ could also manipulate a $B$ without type errors. Subtyping has been discussed at length elsewhere [Bru96, Bru02], so we will not give a general account here.

If $A$ and $B$ are object types, i.e. records of methods, then $B <: A$ if $B$ is an extension of $A$, i.e. if it includes all the methods that $A$ does and possibly more. Therefore, if $A$ is the object type generated by $AClass$ and $B$ the one generated by $BClass$, then we will usually have $B <: A$ if $BClass$ inherits from $AClass$.

However, this breaks in the presence of binary methods. If $AClass$ defines a method $m : MyType \to T$ (for an arbitrary type $T$), then this expands to $m : A \to T$ in $A$ and $m : B \to T$ in $B$. Since $m$ in $B$ cannot accept a parameter of type $A$, objects of type $B$ can no longer masquerade as $A$s in all contexts. Therefore $B \not<: A$.

This is not as catastrophic as it seems. Even though $B$ is no longer truly a subtype of $A$, it is still "almost" a subtype: $B$ has all the same methods as $A$, so code that calls methods of $A$ can call the same methods of $B$, with the caveat that any binary methods can now accept only $B$s and not $A$s. To capture this, we define the *matching* relation, which generalizes subtyping:

**Definition 3.1** (Matching)**.** An object type $B$ *matches* an object type $A$, written $B <\# A$, if every method of $A$ is also found in $B$ with the same signature (up to alpha-conversion).

In this context, `MyType` is considered as a free variable, so binary methods do not prevent types from matching. In $\mathcal{LOOM}$, if a class $AClass$ generates object type $A$ and $BClass$ generates object type $B$, then $B <\# A$ whenever $BClass$ inherits from $AClass$.

### 3.1.4 Match-bounded polymorphism and hash types

There are two primary uses of subtyping in object-oriented languages. The first is to write polymorphic functions, which can accept arguments of a type $A$ or any of its subtypes. The second is to create variables that can store a value of $A$ or any of its subtypes, for instance as elements of a heterogeneous

collection. Despite the fact that `MyType` breaks subtyping, we can actually recover both of these facilities in $\mathcal{LOOM}$, using matching.

Polymorphic functions can be written in $\mathcal{LOOM}$ using *match-bounded polymorphism*. This is akin to the bounded parametric polymorphism available in Java 5, but with a matching relation providing the upper bound rather than a subtyping relation. A polymorphic function takes a type parameter, where the type variable is constrained to match some known type. Then the function can be called on any object type satisfying the constraint, where occurrences of the type variable in the function's signature are substituted with the actual type argument at the call site. For example, see the following listing:

**Listing 3.3** (Match-bounded polymorphism)**.**

```
Base = objecttype
  m1 : proc (integer);
  m2 : proc (mytype);
end;

A = objecttype include Base ... end;
B = objecttype include Base ... end;
AClass = class ... end;   -- Generates objects of type A
BClass = class ... end;   -- Generates objects of type B

procedure f [T <# Base] (x, y : T)
  var a = new AClass   : A;
begin
  x.m1(47);  -- OK, x has m1
  x.m2(y);   -- OK, x has m2 and y is same type as x
  x.m2(a);   -- Error, T is not known to be A
end

var a1 = new AClass  : A;
    a2 = new AClass  : A;
begin
  f[A](a1, a2);
end;
```

Here, `f` is a bounded polymorphic function whose type parameter is constrained to match `Base`. (In $\mathcal{LOOM}$, type parameters and arguments are enclosed in square brackets `[...]` while ordinary parameters and arguments

use parentheses.) Both `A` and `B` match `Base` (the notation `include Base` copies all the method signatures from `Base` into a new object type). Therefore, `f` can accept either `Base`, `A`, or `B` as the argument to its type parameter. In this case, we call `f` with `A` as the type argument. This means that at the call site, `T` (the formal name of the type parameter) is substituted with `A` in the signature of `f`, so the two value parameters `x` and `y` must be passed arguments of type `A`.

It is important to note that the body of `f` is only type-checked and compiled *once*. Calling `f` with type argument `A` does *not* generate a new version of `f` whose body is type-checked under the assumption that `T` = `A`. This is why the third statement in the body of `f` is illegal; when `f` is compiled, we know only that `T` <# `Base`, so the method `m2` will only accept arguments whose type is known to be `T`.

Match-bounded polymorphism can be used to write a function that accepts objects of any type matching $A$. To provide variables that are equally flexible, $\mathcal{LOOM}$ provides *hash types*. If $A$ is an object type, then $\#A$ (read "hash-A") is the type of variables that can store a reference to any object whose type matches $A$. Any of the methods of $A$ can be called on a value of type $\#A$ *except* those including `MyType` as a parameter type. Such calls are not statically type-safe, since the exact type of the object stored in a variable of type $\#A$ isn't known at compile time.

Note that some bounded polymorphic functions could be rewritten as functions taking hash-typed parameters. However, these are less flexible than bounded polymorphic functions, since hash types forbid *all* calls to methods with `MyType` parameters, while bounded polymorphic functions allow these methods to be called so long as the receiver and argument can be established to be the same type, as in the call `x.m2(y)` in Listing 3.3. Moreover, $\mathcal{LOOM}$ provides no way to express the constraint that two hash-typed parameters should have the same run-time type, while bounded polymorphic functions can express this by using the same type variable for both parameters, again as demonstrated in Listing 3.3.

Despite the fact that object types containing methods with `MyType` parameters have no proper subtypes, we can still recover nearly all of the expressiveness that subtyping provides to object-oriented languages, by replacing subtyping with matching. For this reason, $\mathcal{LOOM}$ dispenses with subtyping entirely and uses matching as the sole mechanism for polymorphism and heterogeneous data structures.

## 3.2 Module language

$\mathcal{LOOM}$'s module system was originally designed by Leaf Petersen [Pet96], and was later extended by others including Joe Vanderwaart [BPV98]. It is based to a large extent on the module system of Modula-3. In this section, we will describe the $\mathcal{LOOM}$ module language as it existed before this work began.

### 3.2.1 Interfaces, modules, and programs

In $\mathcal{LOOM}$, a *compilation unit* is either an interface, a module (which provides the implementation for an interface) or a program (which provides the starting point of execution). Compilation units can be in separate source files and can be processed in any order, so long as units are defined before they are used. That is, an interface must be processed before any modules or programs that implement or import the interface; other than this, no restrictions are placed on the order of compilation units. Only one module can implement a given interface at a time, though different implementations can be linked in to produce different complete programs.

An interface consists of a set of declarations and definitions of types, and a set of declarations of constants (which can include constants of primitive types as well as classes and free functions). Types can be declared with no definition, in which case they are *opaque* types, or they can be declared equal to some other type, in which case they are *fully revealed*. Object types declared in an interface can also be *partially revealed*; this is discussed in §3.2.2.

A module definition in $\mathcal{LOOM}$ specifies the interface that it is intended to implement; modules do not have separate names. The body of a module consists of a set of type definitions and constant definitions; these must be consistent with those declared in the interface. All types declared opaque or partially revealed in the interface must be fully defined in the implementation, and all constants declared in the interface must be given a value in the implementation.

A program consists of type definitions and constant definitions which are local to the program, followed by a block, which may declare variables and then execute a sequence of statements. There can be at most one program in a linkset (a list of compilation units to be linked together). The program's block acts as the top-level function and entry point for the complete program.

Both module implementations and programs may also import additional interfaces. When this is done, the names declared by the interface are ac-

cessible in the importing module or program, though only through fully-qualified names, which in $\mathcal{LOOM}$ consist of an interface name, the scope-resolution operator `::`, and a symbol name. For instance, `A::b` refers to the symbol `b` declared in interface `A`.

Interfaces may also import additional interfaces. This form of import works transitively; if interface $A$ imports $B$, then any implementation or program that imports $A$ also gains access to names declared by $B$.

The following listing demonstrates an interface, its implementation, and a program importing the interface:

**Listing 3.4** (Basic modularity)**.**

```
interface Rationals;
  type rational;   -- Opaque type
  make : func (integer, integer)   : rational;
  add  : func (rational, rational) : rational;
  ...
end;

module implements Rationals;
  type
    rational = objecttype ... end;
  const
    class RationalClass ... end;
    function make (n : integer, d : integer) : rational
      begin ... end;
    function add (a : rational, b : rational) : rational
      begin ... end;
    ...
end;

program p;
  import Rationals;
var
  r : Rationals::rational;
begin
  r := Rationals::add(Rationals::make(1, 2),
                      Rationals::make(1, 6));
end;
```

Within the `program` scope, variables of type `Rationals::rational` can

be declared and used, but their representation as an object type with some methods is hidden within the `module` scope.

### 3.2.2 Partial revelation

As seen in the previous section, $\mathcal{LOOM}$ modules provide good information hiding in the form of opaque types. As mentioned in §2.2, modules can produce abstractions in an ADT style by declaring an opaque type and some set of functions that operate on it.

However, ADT-style abstractions can be clunky to use. In an object-oriented language, we really want to be able to define data abstractions that make use of the object model, particularly for the benefits of polymorphism. To do this and maintain good information hiding, we need to be able to define object types for which some methods are visible outside the module, and some are visible only within the module. $\mathcal{LOOM}$ provides partial revelation as a mechanism for this. When an type is partially revealed in an interface, it is declared to match some given object type. The exact type remains unknown, but methods revealed by the matching bound can be called. Partially revealed types are treated very much like type parameters inside a bounded polymorphic function.

The following listing demonstrates partial revelation:

**Listing 3.5** (Partial revelation)**.**

```
interface SetOfInt;
  IntSetType <# objecttype
    add       : proc (integer);
    remove    : proc (integer);
    contains  : func (integer) : bool;
    intersect : proc (mytype);
  end;
  newSet : func () : IntSetType;
end;

module implements SetOfInt;
  type
    TreeNode = objecttype ... end;
    IntSetType = objecttype
      -- These methods are revealed by the interface
      add       : proc (integer);
      remove    : proc (integer);
```

```
      contains  : func (integer) : bool;
      intersect : proc (mytype);
      -- These methods are only visible
      -- in the implementation
      getRoot         : func () : TreeNode;
      recursiveInsert : proc (integer, TreeNode);
      ...
    end;
  const
    newSet = function () : IntSetType  begin ... end;
    ...
end;
```

The interface defines a type `IntSetType`, whose exact definition is unknown outside the module but which is defined to match an object type with methods `add`, `remove`, `contains`, and `intersect`. Within the module, a complete definition of `IntSetType` is given. For a module or program that imports `IntSetType`, the four methods revealed in the interface will be in scope, but the additional methods in the implementation will be inaccessible.

## 3.3 Evaluation of $\mathcal{LOOM}$ modules

How does the original $\mathcal{LOOM}$ module system perform with regard to the features of good module systems discussed in §2.2?

First of all, it is clear that $\mathcal{LOOM}$ modules have excellent support for interface/implementation separation and information hiding. Interfaces are lexically separated from the implementation, so programmers do not need to read the implementation of a module to learn what functionality it exports. With opaque types and partial revelation, interfaces can avoid exposing any more information about their implementations than is necessary. Moreover, this is done in a way that preserves static type safety, and the semantics developed for type-checking match-bounded polymorphic functions in the $\mathcal{LOOM}$ core language can be re-applied to handle partially revealed types. An interpreter supporting the original module system was implemented by Leaf Petersen; though a compiler with modules has not been completed, we are confident $\mathcal{LOOM}$ modules are separately compilabile because the interpreter type-checks each module using only the information present in interfaces imported by the module.

On the other hand, $\mathcal{LOOM}$'s management of namespaces leaves something to be desired. When an interface is imported, the client code must

use fully qualified names to access the imported definitions. While this avoids namespace pollution and accidental name conflicts, using fully qualified names all the time is somewhat cumbersome. It would be useful to follow Java, Python, D, and Modula-3 in adding selective import of names without their fully qualified prefixes, perhaps with optional renaming.

Finally, the modularity features described up to this point support the open/closed principle rather poorly. Since only one interface can be ascribed to each module, programmers must decide up front whether to write a minimal interface, which provides only the information needed for clients to use the module, or write a more generous interface that provides information needed for clients to extend the module. In the former case, extension is difficult since the module is locked up tightly. In the latter case, even ordinary clients are able to see some of the implementation, which seriously weakens information hiding. It is clear that a more expressive module system, allowing modules to treat different clients differently, is needed.

## 3.4 Approaches to multiple-interface systems

The idea of a module that can expose different interfaces to different clients is not new [Fre95, BA99, FF99, Thu02, BAF03]. In this section we will discuss some existing approaches to the problem. Beyond supporting the open/closed principle, multiple interfaces can also be useful in cases where a module interacts with multiple other modules in a large software system. The module could be designed with an interface for each client, each of which was tailored to that client's needs.

### 3.4.1 Opening

Earlier versions of $\mathcal{LOOM}$ included a facility called *opening* [Van97, BPV98]. Opening is a form of import that bypasses the entire information hiding principle and brings all the definitions from a module's *implementation* into the opening scope.[2] The addition of opening improves the situation slightly, since a minimal interface can be written for most clients to use, while extending clients can open the module and get access to the details they need. However, this is still far from desirable, as we may not wish to reveal *all* the implementation details to any client.

[2]Note that this differs from the definition of opening in Standard ML, which simply means that imported symbols are accessible in short form, without module-name prefixes.

### 3.4.2 Monotonic interfaces

A simple modification of opening is the idea of monotonic interfaces, in which we have a sequence of interfaces, each revealing strictly more than its predecessor. For example, a module could have a minimal interface for normal clients and a second, more generous interface that reveals everything in the minimal interface as well as additional information useful for extending clients. More than two interfaces can also be supported in this paradigm.

Monotonic interfaces are easy to type-check, since the union of any combination of these interfaces is simply the biggest (most generous) one in the list. Therefore, type-checking reduces to the single-interface case together with checking that each interface includes all the revelations from its predecessor. However, monotonic interfaces still do not provide all the expressiveness we might desire: they do not allow us to provide completely distinct interfaces, which can be useful for a module with different clients that use different parts of its functionality.

### 3.4.3 General multiple interfaces

What we would really like to allow is for every module to have a completely arbitrary set of interfaces associated with it, each of which is independent of the others and reveals precisely the subset of the module's functionality that the interface designer desires. There should be no restrictions on how many interfaces a module may have or what the relationship between those interfaces may be; a module's interfaces may be totally disjoint or they may overlap, and they may be monotonic, or not.

This approach has consequences for type-checking. A module's interfaces obviously must be consistent with its implementation. However, for static type checking and separate compilation to work, clients must be able to determine whether a set of interfaces is self-consistent without reference to their common implementation. If we had access to the implementation, we could simply check that each declaration in each interface was consistent with its corresponding full definition (in the implementation); these definitions would then be the ultimate arbiters of correctness. However, a client must be able to type-check and compile *without* seeing the complete definitions of the symbols it imports. This necessitates a more sophisticated algorithm capable of assessing the internal consistency of a set of interfaces. It must be an error for one interface to contradict another.

If we take the information-hiding principle to its logical conclusion, clients should not even know that a set of interfaces shares a common im-

plementation. If they did, it would compromise the freedom that strong information hiding is supposed to provide: that we ought to be able to seamlessly and transparently swap implementations conforming to the same interface(s) at link time, without recompiling client code. In particular, if a program imports two interfaces, we should be able to link it either with two separate modules (one for each interface) or with a single module implementing both interfaces.

Diane Bennett points out a problem that this approach raises [Ben03]: interfaces may introduce *aliasing*, artificial distinctions between types. If a client imports two interfaces with a common implementation, one would expect to be able to access the union of the functionality exposed by both interfaces. However, if the same underlying object type is partially revealed in each interface, a naïve implementation will treat the two declarations as independent, and a client will not be able to use objects created from one interface with the functionality from the other interface. Examples of this phenomenon will be discussed in §3.5. We see that the goals of expressiveness and of type-safe separate compilation are at odds here.

In $\mathcal{LOOM}$ we compromise by introducing the means for an interface to reveal that a type shares an implementation with a type from another interface. With this extra information, clients are able to use separate declarations of the same underlying type interchangeably. This does weaken information hiding slightly, but if done carefully it reveals the minimum amount of information necessary to allow the increased expressiveness we desire.

## 3.5 Examples using multiple interfaces

To make the ideas in the preceding discussion more concrete, we present two example programs demonstrating what we would like to be able to do with the $\mathcal{LOOM}$ module language.

### 3.5.1 Extending a closed-source module

This example is adapted from [Ben03], and considers the following problem: suppose we have a closed-source module implementing a `Widget` class, with two interfaces: a minimal one, which reveals only the methods needed for an application to use the widget, and a more generous one for clients who want to subclass the widget to create their own kinds of widget. Such interfaces might be defined as follows:

**Listing 3.6** (A module with two interfaces)**.**

```
interface Widget;
  -- Partially reveal an object type for the widget
  Widget <# objecttype
    redraw: proc ();
    ...
  end;

  -- A factory function to let clients create widgets
  makeWidget: func () : Widget;
end;

interface WidgetFull;
  -- Fully revealing the widget's object type
  Widget = objecttype
    -- The same methods we saw before
    redraw: proc ();
    ...
    -- Also reveal some internal methods
    drawBorder:   proc ();
    drawContents: proc ();
    ...
  end;

  -- A class generating Widgets, which clients
  -- can subclass
  WidgetClass: classtype ... end;
end;
```

Note that the minimal interface reveals a factory function rather than a class for creating `Widget`s, thus hiding any fields and implementation methods the underlying class may have (since if we revealed a class, we would have to reveal its type, which contains full information about all fields and methods). The generous interface, however, must reveal the class since anyone who wants to extend the widget must have access to the class in order to inherit from it.

Now, suppose we wish to create a new version of this widget which has a border only on Sundays, and not on the other days of the week. Naturally, we would write a module that defines an extension of `Widget` overriding its

`redraw` method with our customized functionality. We might attempt to do this as follows:

**Listing 3.7** (Extending the `Widget` module)**.**

```
interface CustomWidget;
  import Widget;

  -- Declare a new object type that matches
  -- the original Widget.
  CustomWidget <# Widget::Widget;

  -- A new factory function.
  makeCustomWidget: func () : CustomWidget;
end;

module implements CustomWidget;
  import Widget, FullWidget;

  class CustomWidgetClass
    inherit FullWidget::WidgetClass
    modifying redraw;
  methods
    procedure redraw ()
    begin
      -- Draw the border only if it's Sunday
      if getDayOfWeek() = Sunday then
        drawBorder();
      end;
      drawContents();
    end;
  end;

  function makeCustomWidget () : CustomWidget
  begin
    -- The following line triggers a type error
    return new(CustomWidgetClass);
  end;
end;
```

Unfortunately, this code will not compile. The problem is that from the application's point of view, `Widget::Widget` and `WidgetFull::Widget` are

completely different types; there is no relationship between them. Therefore, when we try to type the result of `new(CustomWidgetClass)` as `CustomWidget`, the type-checker will report an error since `CustomWidget` is declared to match `Widget::Widget` but `CustomWidgetClass` is implemented in terms of `WidgetFull::Widget`.

Why is it necessary to state in the interface that `CustomWidget` matches `Widget::Widget`? This is required if we want to be able to use `CustomWidget`s like `Widget`s. For instance, suppose we had a polymorphic function with a type parameter `T <# Widget`, or a list whose elements are of type `#Widget`. In either case, we would expect to be able to use `CustomWidget`s interchangeably with `Widget`s, but this is only possible if the type-checker judges `CustomWidget` to match `Widget`. Since both types are partially revealed, this constraint must be declared in the interface; it cannot be inferred from outside the module implementing `CustomWidget`.

To allow `Widget` to be extended, the designer of the original module would have to add the following lines in the `WidgetFull` interface:

```
import Widget;
Widget = Widget::Widget;
```

This causes the type-checker to identify the `Widget` type being declared (which is `WidgetFull::Widget` since we are in the `WidgetFull` interface) with `Widget::Widget`, the type declared in the minimal `Widget` interface. With this addition, the extension module shown above will work.

Complete source code for this example can be found in §C.1.

### 3.5.2 Providing distinct functionality to different clients

In the previous example, we considered what is necessary to extend a module without altering its implementation. Another case where multiple interfaces can be handy is when a module interconnects with multiple other components in a large software system. Such a module could be designed with an interface specialized for each component it faces.

As an example, let's consider a module in a student information system for a university. The system stores information about students in a database and provides varying levels of access to that information to different kinds of users. The module we consider acts as an intermediary between the database back-end and a front-end application; it provides an object-oriented view of the database contents. In particular, it is responsible for providing a `Student` object type. An application can look up a student record in the

database and get a reference to the corresponding `Student` object; methods of this object can then be called to retrieve information about the student, such as his or her name, GPA, financial aid status, and so on.

Let's suppose that varying levels of access are granted to different users by providing several interfaces to our module, each of which exposes only methods that retrieve information the client is permitted to see.[3] We'll consider two classes of users: registrars and financial aid officers. Registrars should be able to see a student's class schedule and GPA, but not any financial aid-related information; financial aid officers should be able to see the student's income and loans, but not any schedule or grade information. However, there are some pieces of information both will be able to access, such as the student's name and permanent address. Therefore, we might write the following interfaces:

**Listing 3.8** (Overlapping interfaces)**.**

```
interface Registrar;
  -- Declare a Student type, revealing only the
  -- information registrars can see
  Student <# objecttype
    getName: func () : string;
    getAddress: func () : string;
    getGPA: func () : real;
    getCurrentSchedule: func () : List[CourseSection];
    ...
  end;

  -- Look up a student by ID number
  getStudent: func (integer) : Student;
end;

interface FinancialAid;
  -- The same Student type, but revealing only the
  -- information financial aid officers can see
  Student <# objecttype
    getName: func () : string;
    getAddress: func () : string;
    getIncome: func () : real;
```

[3] Of course, a *real* student information system should not be designed this way; it is brittle and probably insecure. However, let's ignore that for the purposes of this example.

```
    getLoans: func () : List[StudentLoan];
    ...
  end;

  -- Look up a student by ID number
  getStudent: func (integer) : Student;
end;
```

Here, a similar problem arises to the one we saw in the previous example. Although there may be a single underlying implementation module with one `Student` type that is revealed by both interfaces, the interfaces do not contain the information that the `Student` from `Registrar` is interchangeable with the `Student` from `FinancialAid`. This means that if a single client imports both interfaces, it will find that a `Student` obtained via the `Registrar` interface cannot then be used with the `FinancialAid` interface:

**Listing 3.9** (Importing overlapping interfaces)**.**

```
program StudentTest;
-- By importing both interfaces, we should have access
-- to the union of the Student methods from each.
import Registrar, FinancialAid;
var
  theStudent: Registrar::Student;
  GPA, income: real;
begin
  -- Look up some student
  theStudent := Registrar::getStudent(...);

  -- Get the student's GPA...we can do that
  GPA := theStudent.getGPA();

  -- Get the student's income...oops, type error
  income := theStudent.getIncome();
end;
```

The call to `getIncome` fails because `theStudent` is typed as `Registrar::Student`, which is not judged equal to `FinancialAid::Student` by the type checker, and therefore does not have the `getIncome` method. The solution, as before, is to add the equality information to the original interfaces. To do this, we define the `Student` type in a third interface (let's call it

`InfoSys`), which reveals only the information about `Student` that everyone has permission to access. Both the `Registrar` and `FinancialAid` interfaces then import `InfoSys` and declare their own `Student` types to be equal to `InfoSys::Student`. This allows the program in the preceding listing to compile (note that the program does *not* need to explicitly import `InfoSys`). This technique is demonstrated in the following listing.

**Listing 3.10** (Factoring out `Student` to a third interface)**.**

```
interface InfoSys;
  -- Declare a student type, revealing only the
  -- information everyone can see
  Student <# objecttype
    getName: func () : string;
    getAddress: func () : string;
  end;

  -- Look up a student by ID number
  getStudent: func (integer) : Student;
end;

interface Registrar;
  import InfoSys;
  -- Reveal additional registrar-only information
  Student = InfoSys::Student;
  Student <# objecttype
    getGPA: func () : real;
    getCurrentSchedule: func () : List[CourseSection];
    ...
  end;
end;

interface FinancialAid;
  import InfoSys;
  -- Reveal additional financial-aid-only information
  Student = InfoSys::Student;
  Student <# objecttype
    getIncome: func () : real;
    getLoans: func () : List[StudentLoan];
    ...
  end;
end;
```

Why is it necessary to create a third interface? We could, if we weanted, have `FinancialAid` simply import `Registrar` and declare `Student = Registrar::Student`. However, this would have the side effect of revealing all the registrar-related functionality to anyone who imported `FinancialAid`, which is undesirable. Therefore, both `Registrar` and `FinancialAid` must link their `Student` types to `InfoSys::Student`, which reveals only the information that both registrars and financial aid officers are allowed to see.

The full source code for this example can be found in §C.2.

We have now outlined how a module system allowing general multiple interfaces ought to work. In the following chapter we will examine in detail how a type-checker for such a module system can be implemented.

# Chapter 4

# Multiple-interface modules

In the previous chapter, we outlined the features of a general multiple-interface module system and gave examples of how programmers might use such a system. In this chapter, we will discuss how this system is actually implemented in our prototype $\mathcal{LOOM}$ type-checker.

## 4.1 Overview

We have implemented a general multiple-interface module system, as described in §3.4.3, in the $\mathcal{LOOM}$ type-checker. Our system allows for a module to have an arbitrary set of interfaces, while interfaces can be type-checked and used in client code without access to their implementations. To solve the aliasing problem discussed in §3.4.3, we allow interfaces to specify equality constraints between types; for instance, an interface $A$ can include a declaration $T = B::T$, where $B$ is another interface, to specify that the type-checker should assume the types $A::T$ and $B::T$ to be equal. These equality constraints are combined with the matching constraints used in partial revelation (see §3.2.2) in a single framework: a constraint graph. We will discuss constraint graphs further in §4.3.

## 4.2 Syntax

In the new module system, interfaces contain declarations that introduce type names and impose constraints on types. The constraints that can be introduced are of two kinds: matching and equality constraints. The former asserts that some type $S$ must match a type $T$, and the latter that a type $S$ equals a type $T$.

For example, the `WidgetFull` interface (see Listing 3.6, page 26) in its final form contains two constraints.

**Listing 4.1** (Syntax of an interface)**.**

```
interface WidgetFull;
  import Widget;
  Widget <# objecttype       -- (1)
    redraw: proc ();
    ...
  end;
  Widget = Widget::Widget;   -- (2)

  makeWidget: func () : Widget;
end;
```

The constraints are (1) that `Widget` matches an `objecttype` with the `redraw` method, and (2) that `Widget` is equal to `Widget::Widget`. This interface also declares the constant `makeWidget`, which happens to be a function.

Note that in declarations of type constraints, a name on the left side of the constraint can declare a new type. In the preceding example, the type `Widget` is declared by the line marked (1). Line (2) then refers, of course, to the same `Widget` type; there cannot be two types with the same name within the same scope. A name on the right side of a constraint must refer to a previously declared type that is visible in the current scope.

The syntax for a module implementing multiple interfaces is simply to list all the interfaces in the preamble of the module:

**Listing 4.2** (Syntax of an implementation module)**.**

```
module implementing Widget, FullWidget;
  type
    Widget = objecttype ... end;
    ...
  const
    class WidgetClass ... end;
    function makeWidget () : Widget ... end;
    ...
end;
```

As before, every type declared in any of the interfaces being implemented (that is, appearing on the left side of a matching or equality constraint) must be given a full definition in the module body. The only exception to this rule

is for types that are fully revealed in one of the interfaces (that is, involved in an equality constraint whose right side is an explicit `objecttype`). The module body is not required to restate the definition of fully-revealed types.

## 4.3 Constraint graphs

When we type-check a $\mathcal{LOOM}$ module or program, one of the first things we must do is collect the interfaces that it imports and accumulate the information stored in them into some sort of data structure. When type-checking the body of the module or program, we will often refer to this data structure, e.g. to type-check a method call on an object whose type is partially revealed in an imported interface. The representation we have chosen to accumulate the information about types revealed by interfaces is a *constraint graph*.

In this graph, the nodes stand for types; some are *named types*, identified by a name declared in some interface (which may be opaque, partially revealed, or fully revealed); others are *explicit types*, which are of the form `objecttype ... end`, i.e. they contain an explicit listing of all an object type's methods. Object types are the only kind of types in a constraint graph. Any named type declared in an interface is assumed to stand for some object type, since object types are the only ones that can be involved in matching relations.

The edges of the graph are constraints between the types. Edges are labeled with the kind of constraint they represent, either "$<\#$" for matching or "$=$" for equality. Matching edges are directed, while equality edges are considered to be bidirectional. A matching edge from $A$ to $B$ means that $A <\# B$.

For example, the `Widget` interface shown in the previous section would give rise to the constraint graph shown in Figure 4.3.

As discussed in §3.4.3, when we type-check a group of interfaces, we must ensure that their declarations are self-consistent. Here, the requirement of consistency becomes a condition on the constraint graph. We could check each declaration of each interface as it is inserted into the graph to ensure that it is consistent with the information already in the graph, but instead we have chosen to implement a scheme in which constraints are inserted with only minimal checking, then the whole graph is checked for consistency once at the end. The reason for this is that the constraint graph constructed from a group of interfaces might have a complicated structure that would make it difficult to make equality and matching queries; that is, it would

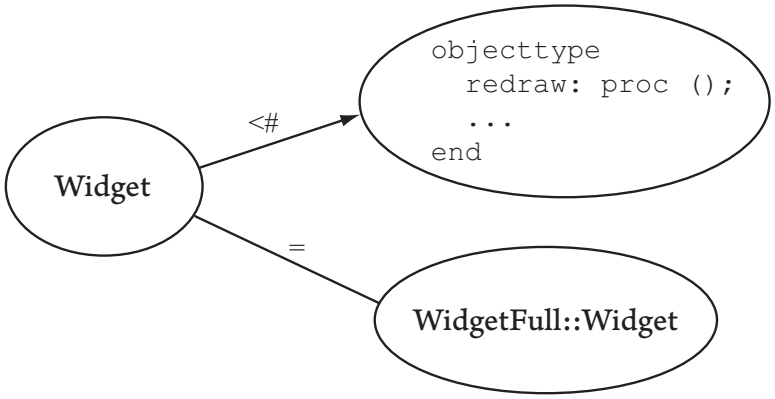


Figure 4.3: The constraint graph resulting from the `Widget` interface.

be difficult to use the graph to answer questions of the form "does $A$ match $B$?" or "is $A$ equal to $B$?" when $A$ and $B$ are types involved in constraints. Therefore, before using the graph for type-checking a module body, it is *regularized*. By this we mean that the complicated "raw" constraint graph is transformed into an equivalent graph with a simple structure that makes equality and matching determination easy and efficient. Conveniently, it turns out to be easy to check the graph for consistency at the same time it is regularized. The following sections will describe the regularization and consistency checking algorithms in more detail.

To summarize, the outline of the algorithm for type-checking a module is as follows. (A program can be treated as a module that does not implement any interfaces and that contains top-level code serving as the entry point.)

**Algorithm 4.4** (Module type-checking)**.** For each module:

1. Find all the interfaces imported by the current module (including those imported indirectly via transitive import).
2. Add all constraints from the imported interfaces to an initially-empty constraint graph.
3. Add all constants[1] defined by the imported interfaces to an initially-empty type environment.

[1]Recall that constants in $\mathcal{LOOM}$ include functions and classes, as well as constant values of primitive types.

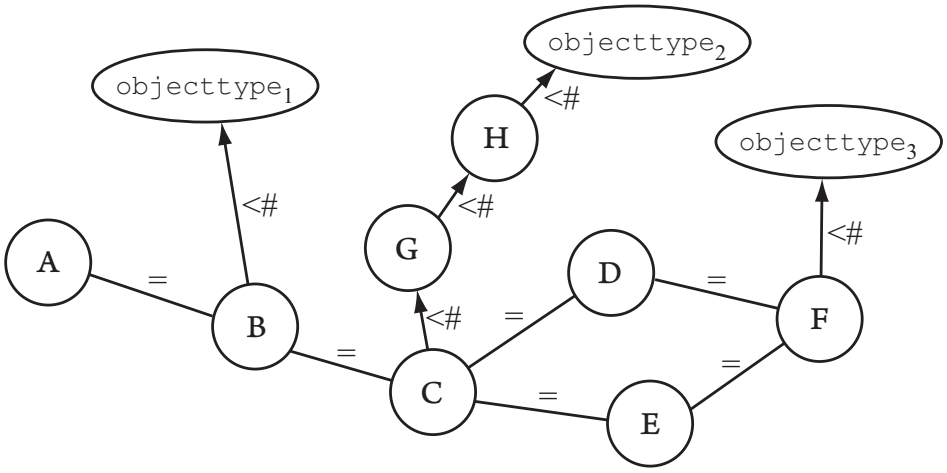


Figure 4.5: In an unregularized constraint graph, equality and matching checking require a full graph search.

4. Regularize the graph and signal an error if it is inconsistent.

5. If the module implements one or more interfaces:

   (a) Check that each type declared in the interfaces being implemented has a definition in the body of the module that is consistent with the constraints placed on it by the interfaces. Add these definitions to the constraint graph and re-regularize it.

   (b) Check that each constant declared in the interfaces being implemented has a definition in the body of the module that is consistent with its declared type in the interfaces.

6. Type-check the body of the module.

### 4.3.1 Regularization

There are two ways in which the "raw" graph produced directly from the constraints in the imported interfaces is too complicated to use directly. One relates to equality queries and the other to matching queries.

First, a type may be provably equal to many more types than its immediate neighbors (via equality edges) in the graph. For instance, in the

constraint graph shown in Figure 4.5, the types $A$ and $F$ are equal, but to determine this, the type-checker would have to search for a path between them via equality edges, using a graph search algorithm such as BFS or DFS. These take time linear in the number of types involved in an equality-connected component of the graph. We would rather not have to pay this cost every time we want to determine if two types are equal.

Similarly, a matching query might require a full graph search. Figure 4.5 shows that $C <\# G$ and $G <\# H$; since matching is transitive, this means $C <\# H$ as well, and so $A <\# H$ (since $A = C$). Determining that $A <\# H$ requires finding a path from $A$ to $H$. Moreover, checking that type $A$ has a particular method may require us to find all the explicit types $A$ matches in the graph (in this case, $\texttt{objecttype}_1$, $\texttt{objecttype}_2$, and $\texttt{objecttype}_3$) to locate the method.

The goal of regularization is to transform the graph into one that is equivalent with respect to the constraints but for which equality and matching determinations can be made efficiently. This is done in three major phases.

*Phase 1.* In the first phase we partition nodes into equivalence classes based on equality. For each equivalence class, we choose a single *representative node*. If the class contains an explicit type, then that type becomes the representative; otherwise, a named type is chosen arbitrarily. All other nodes in the equivalence class are kept in the graph with just a single equality edge to the representative, and all matching constraints on nodes in the equivalence class are moved to the representative.

Equivalence classes are determined by simply doing a BFS, starting from an arbitrarily chosen node, which follows only equality edges. All nodes reached by the BFS are therefore part of the same equivalence class.

Note that a cycle involving matching edges could also prove equality. For instance, if $A <\# B$ and $B <\# C$ and $C <\# A$, then $A = B = C$. Our implementation does not currently check for such cycles, on the assumption that it is unlikely such patterns would be created accidentally by a programmer. However, if necessary, these cycles could be detected straightforwardly using a BFS/DFS that followed matching edges as well as equality edges.

Figure 4.6 shows the example graph from Figure 4.5 after the first phase of regularization. Note that equality edges connect several types directly to $B$, chosen as the representative of its component, and that all matching constraints on types in that component have been moved to $B$.

At this point, equality determination is simple. For each type to be compared, the representative of its component is found; the representatives are then compared. For example, to determine that $A$ and $G$ are equal, we

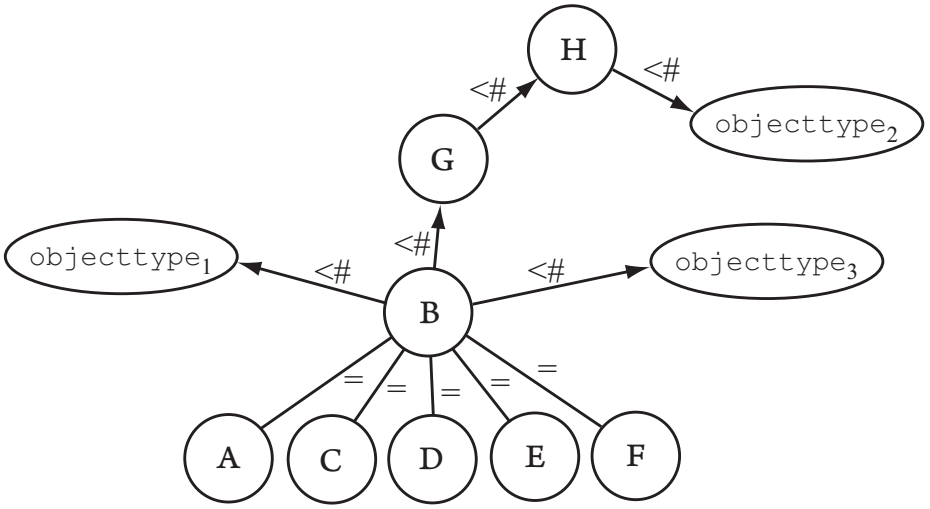


Figure 4.6: After the first phase of regularization.

look up their representatives, which are both $B$. Choosing an explicit type (if one is available) as the representative ensures that equality queries also work when one of the parameters to the query is an explicit type.

*Phase 2.* In the second phase, we propagate matching constraints "down" the graph, i.e. against the direction of the matching edges. As discussed earlier, matching is transitive, so e.g. $B <\#$ `objecttype`$_2$ in Figure 4.6. We would therefore like $B$ to have a matching edge directly to `objecttype`$_2$, and more generally, every representative node in the graph should have matching edges directly to all the other representative nodes that it matches. This is accomplished by doing a backwards DFS, starting at each "top" node (those without outgoing matching edges) and following the matching edges backwards. At each node encountered, matching edges are added between the node and all its ancestors in the DFS tree, i.e. all nodes it matches via transitivity.

Figure 4.7 shows our example graph after the second phase of regularization; now $B$ has matching edges directly to $H$ and to `objecttype`$_2$, and $G$ also has a matching edge directly to `objecttype`$_2$.

*Phase 3.* The final phase of regularization merges multiple matching constraints to explicit types on a single node. For example, in Figure 4.7, $B$ matches three different `objecttype`s. However, matching several different

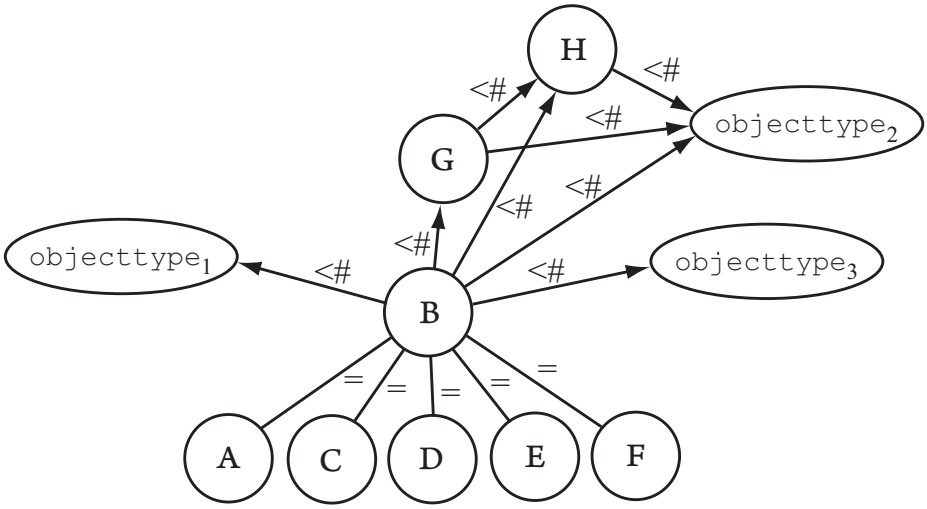


Figure 4.7: After the second phase of regularization.

object types simultaneously is equivalent to matching their greatest lower bound. Note that for matching, the greatest lower bound of a set $S$ of object types is simply the object type with the union of all methods from types in $S$.[2] Therefore, we identify representative nodes that match more than one explicit type, find the greatest lower bound of those types, and replace the several constraints by a single one. Figure 4.8 shows our example graph after this phase. The three `objecttype` nodes have been merged into one, denoted $\texttt{objecttype}_{1+2+3}$, and $B$ now has a single matching edge to that node instead of three separate ones. However, $B$'s matching edges to the partially revealed types $G$ and $H$ are maintained, as are $G$ and $H$'s own matching edges to $\texttt{objecttype}_2$.

After regularization has completed, matching determination is as simple as equality determination. To check whether two named types match, their representatives are found and we check whether there is a matching edge between them. To check whether a named type $A$ matches an explicit type $C$, we look at $A$'s representative and see if it has a matching edge to some explicit type $B$; if so, we check whether $B <\# C$ (by directly comparing their lists of methods, since they are both explicit).

[2]Lower bounds do not necessarily exist for all $S$; see the next section.

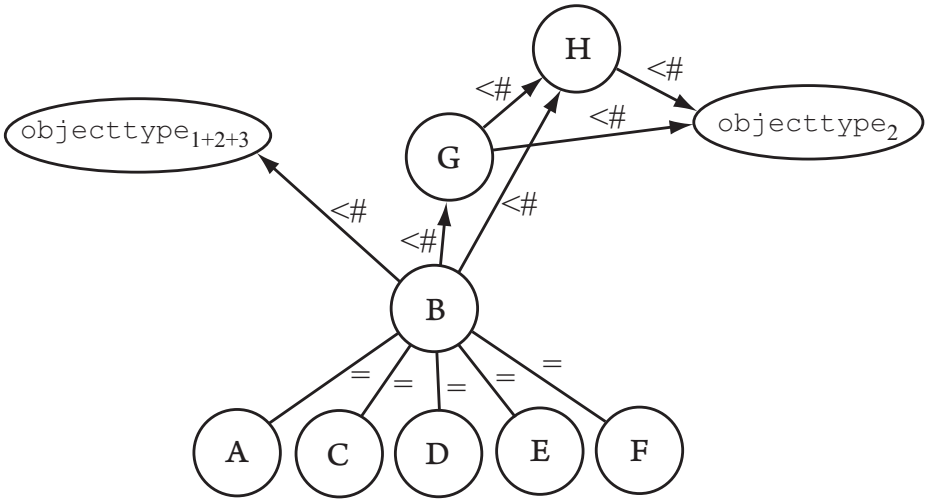


Figure 4.8: After the third phase of regularization.

To summarize, the regularization algorithm works as follows.

**Algorithm 4.9** (Constraint graph regularization)**.**

1. Restructure equivalence classes of equal nodes.
   While unvisited named nodes are left in the graph:
   (a) Pick an arbitrary unvisited named node.
   (b) Do a BFS on equality edges to find the equality-connected component to which it belongs.
   (c) Choose a representative from among the nodes found; if an explicit type was found, choose it, otherwise choose an arbitrary named type.
   (d) Move all matching constraints on nodes in the component to the representative.
   (e) Replace the equality edges within this component by a single edge from each non-representative node to the representative.
2. Propagate matching constraints down the graph.
   (a) Determine the "top" nodes, i.e. those without any outgoing matching edges.

(b) For each top node, do a DFS going backward along matching edges; at each node encountered during the DFS, add matching edges to all its ancestors in the DFS tree.

3. Merge multiple explicit matching bounds on named types.
   For each named node:

   (a) Find all explicit types this node is constrained to match.

   (b) Merge them into one type and replace the existing constraints to explicit types with one to the merged type.

In the next section we will discuss how consistency checking can be incorporated into this algorithm.

### 4.3.2 Consistency checking

By definition, type equality is an equivalence relation and matching is a partial order. For a set of constraints among types to be self-consistent, the constraints should form a model of these relations. That is, the equality relation implied by the graph should obey the axioms of an equivalence relation (which are reflexivity, symmetry and transitivity), and the matching relation implied by the graph should obey the axioms of a partial order (which are reflexivity, antisymmetry and transitivity).

As one can easily convince oneself, these conditions are essentially guaranteed by the design of the regularization algorithm described in the previous section. However, this is actually not quite sufficient to guarantee the consistency that we want. Equality cannot be *any* equivalence relation, and matching cannot be *any* partial order; equality must mean that object types have the same methods with the same signatures up to alpha-conversion, and matching must mean that the upper type has a subset of the lower type's methods. Therefore, the connection between equality and matching in the constraint graph and equality and matching as generally defined by $\mathcal{LOOM}$'s semantics is located wherever *explicit* types are found in the constraint graph.

With this in mind, we see that there are two places in the regularization algorithm where we must introduce additional checks to ensure the equality and matching relations implied by the graph are consistent with $\mathcal{LOOM}$'s semantics. First, in phase 1, when we are collecting nodes into equivalence classes, if two or more explicit types are found in the same equivalence class, we must ensure they are equal up to $\alpha$-conversion before continuing; if they are not, an error is signaled.

Second, in phase 3, when we merge multiple matching constraints to explicit types, we must ensure that the object types being merged are compatible with one another. Recall that merging a set of object types can be described as finding the greatest lower bound for the types being merged, with respect to the matching relation. However, since method names cannot be overloaded in $\mathcal{LOOM}$, if two object types each contain a method of the same name but with different (not $\alpha$-equivalent) signatures, then they have no common lower bound; no object type can contain both methods. Therefore, the merging process looks for method names in common between the types to be merged and accepts them only if same-named methods have the same signature up to $\alpha$-conversion. Otherwise, an error is signaled.

In our implementation of the $\mathcal{LOOM}$ type-checker we have added these two checks to the regularization routines. Though we will not give a formal proof here, the regularization algorithm with these checks should correctly enforce that type declarations from interfaces imported into a module are consistent with $\mathcal{LOOM}$'s semantic rules.

# Chapter 5

# Conclusion

In §3, we identified a major shortcoming in the original module system of $\mathcal{LOOM}$: that it does not provide good support for the principle that a programmer should be able to extend a well-designed module without modifying the code in it. We suggested that the most straightforward way to rectify this problem was to allow for a module to expose multiple interfaces to the outside world; different interfaces could be targeted to different kinds of clients. In §4, we showed how to implement a type-checker for a general multiple-interface module system with no restrictions on the interfaces exported by a module except the basic requirement that they be consistent with one another and with the definitions in the module's implementation.

As a result, we believe that $\mathcal{LOOM}$ with our extended module system now supports the open/closed principle well. Both of the multiple-interface examples in §3.5 (for which full source code appears in Appendix C) can be type-checked under our system, and we believe our system presents no barrier to static type safety or separate compilation. Our system therefore makes $\mathcal{LOOM}$ much more usable for programming in the large than it previously was.

Much work remains to be done on this subject, both theoretical and practical. On the theoretical side, we have not attempted to prove that type-checking is sound under our new semantics, and we have not attempted to formally prove separate compilability (though the language is certainly separately compilable, since the type-checker uses information from interfaces only when importing a module). Separate compilability might be proved using a formal semantics of linking like that developed by Cardelli [Car97]. In addition, it would be useful to investigate the complexity of type-checking and of our regularization algorithm (Algorithm 4.9 on page 41); we believe the latter to be at worst quadratic in the size of a single connected compo-

nent of the constraint graph, and linear in the number of such components. Given typical programming practice, it may well be the case that the size of a connected component is constant-bounded in real-world cases, making regularization effectively linear in the number of types declared by imported interfaces.

Additionally, we would like to investigate the possibility of generalizing $\mathcal{LOOM}$'s module system still further to include a parameterized-import system akin to the "units and mixins" of Findler and Flatt [FF99], which we discussed briefly in §2.4. Findler and Flatt's solution to the "expression problem" is particularly attractive, and it would be interesting to see if this could be realized in $\mathcal{LOOM}$ while maintaining static type safety.

On the practical side, although we have implemented a type-checker for our extension of $\mathcal{LOOM}$, we do not yet have an interpreter for it; our front-end code base had fallen into serious disrepair and we decided to rewrite it almost entirely at the beginning of this project, which unfortunately means the interpreter will need a significant amount of work before it will become functional again. Moreover, our type-checker leaves out "typegroups", a feature introduced in $\mathcal{LOOM}$ in [BV99] (following similiar work done with an extension of Java; see [Bru97]). Typegroups, like modules, group together a set of related definitions; however, the primary purpose of typegroups is to allow programmers to define mutually-recursive groups of object types that can then be extended as one, while the purpose of modules is to provide abstraction barriers, information hiding, and separate compilation. We therefore believe typegroups and modules are orthogonal features, although their interaction has not been fully investigated.

# Bibliography


[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Prentice Hall, fourth edition, 2005.

[BA99] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847, 1999.

[BAF03] Lujo Bauer, Andrew W. Appel, and Edward W. Felten. Mechanisms for secure modular programming in Java. *Software Practice and Experience*, 33:461–480, 2003.

[BCC+95] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[Ben03] Diane Bennett. Aliasing in $\mathcal{LOOM}$. Unpublished monograph, Williams College, September 2003.

[BFP97] K. B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good "match" for object-oriented languages. In *Proc. ECOOP '97*, pages 104–127, 1997.

[BPV98] K. B. Bruce, L. Petersen, and J. Vanderwaart. Modules in $\mathcal{LOOM}$: Classes are not enough. Technical report, Williams College, 1998.

[Bri07] Walter Bright. The D programming language. Online documentation, `http://www.digitalmars.com/d` (accessed 10 Nov 2007), 2007.

[Bru96] K. B. Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. Technical report, Williams College, June 1996.

[Bru97] K. B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Williams College, October 1997.

[Bru02] K. B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[BV99] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Fifteenth Conference on the Mathematical Foundations of Programming Semantics*, 1999.

[Car97] L. Cardelli. Program fragments, linking, and modularization. In *Proc. 24th Annual ACM Symposium on Principles of Programming Languages*. ACM Press: New York, NY, 1997.

[Eif06] ECMA-367—Eiffel: Analysis, design, and programming language. ECMA standard, available online at `http://www.ecma-international.org/publications/standards/Ecma-367.htm` (accessed 10 Nov 2007), June 2006.

[FF99] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34, pages 94–104, 1999.

[FR02] K. Fisher and J. Reppy. Inheritance-based subtyping. *Information and Computation*, 177(1):28–55, 2002.

[FR03] K. Fisher and J. Reppy. Object-oriented aspects of Moby. Technical Report TR-2003-10, University of Chicago, 2003.

[Fre95] Steve Freeman. Partial revelation and Modula-3. *Dr. Dobb's Journal*, 20(10):36–42, October 1995.

[McC04] Steve McConnell. *Code Complete*. Microsoft Press, second edition, 2004.

[Mey97] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, Upper Saddle River, NJ, USA, second edition, 1997.

[Nel91] Greg Nelson, editor. *Systems programming with Modula-3*. Prentice Hall, Upper Saddle River, NJ, USA, 1991.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Pet96] L. E. Petersen. A module system for $\mathcal{LOOM}$. Bachelor's thesis, Williams College, 1996.

[Str00] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, third edition, 2000.

[Szy92] C. A. Szyperski. Import is not inheritance—why we need both: Modules and classes. *Proc. ECOOP '92*, 615:19–32, 1992.

[Thu02] Douglas Thunen. Modules in $\mathcal{LOOM}$ and their separate compilation. Bachelor's thesis, Williams College, 2002.

[Van97] Joe Vanderwaart. The new guide to $\mathcal{LOOM}$. Technical report, Williams College, 1997.

[Wad98] Philip Wadler. The expression problem. Post to Java Genericity mailing list, available online at `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt` (accessed 11 Nov 2007), 12 November 1998.

# Appendix A

# $\mathcal{LOOM}$ module grammar

In this appendix, we give an updated formal grammar for the new $\mathcal{LOOM}$ module system with multiple interfaces. The grammar for the core language can be found in [Pet96].

Nonterminal symbols are set in *italic* while terminal symbols are set in `typewriter text`. (We will also place quote marks around single-character terminals, since it can be difficult to tell whether they are set in typewriter text or not.) The notation ID represents any identifier token. Note that $\mathcal{LOOM}$ is case-insensitive with respect to both keywords and identifiers.

For brevity we will use EBNF notation. $A^*$ denotes zero or more repetitions of $A$, while $A?$ denotes an optional $A$.

**Table A.1** ($\mathcal{LOOM}$ module grammar)**.**

| | | |
|---|---|---|
| *Start* | ::= | *compUnit* (“`;`” *compUnit*)* “`;`”? *EOF* |
| *compUnit* | ::= | *programUnit* \| *interfaceUnit* \| *implementationUnit* |
| *programUnit* | ::= | `program` ID “`;`” *importDecl*? *abbrevList*? *unitBlock* |
| *interfaceUnit* | ::= | `interface` ID “`;`” *importDecl*? *declList*? `end` |
| *implementationUnit* | ::= | `module implements` ID (“`,`” ID)* “`;`” *importDecl*? *abbrevList*? *constList*? `end` |
| *importDecl* | ::= | `import` ID (“`,`” ID)* “`;`” |
| *declList* | ::= | *decl* (“`;`” *decl*)* “`;`”? |
| *decl* | ::= | ID (“`,`” ID)* “`:`” *typeExp* \| ID “`<#`” *typeExp* (“`,`” *typeExp*) \| ID “`=`” *typeExp* (“`,`” *typeExp*) |

| | | |
|---|---|---|
| *typeExp* | ::= | *longName* |
| | | \| `objecttype` *includeDecl*? *methodTypeList*? `end` |
| | | \| ... |
| *includeDecl* | ::= | `include` *longName* (“`,`” *longName*)* “`;`”? |
| *methodTypeList* | ::= | *methodType* (“`;`” *methodType*)* “`;`”? |
| *methodType* | ::= | ID (“`,`” ID)* “`:`” *funcType* |
| *longName* | ::= | ID (“`::`” ID)* |

The rules for *abbrevList*, *constList*, *unitBlock*, and *funcType*, as well as the complete rule for *typeExp*, may be found in §A.2 of [Pet96]. (We have here omitted the cases of *typeExp* that are not relevant to module type-checking, such as primitive types, class types, and so on.)

# Appendix B

# Type-checking rules for $\mathcal{LOOM}$ modules

In this appendix, we give updated formal type-checking rules for the new $\mathcal{LOOM}$ module system with multiple interfaces. Formal type-checking rules for the core language can be found in [Pet96].

**Definition B.1** (Type constraint systems)**.**

1. If $\sigma$ is a named type and $\tau$ is either a named type or an explicit `objecttype` with no free variables, then the statements $\sigma = \tau$ and $\sigma <\!\!\# \tau$ are type constraints.

2. A type constraint system $\mathcal{C}$ is a set of type constraints. $\mathcal{C}_0 \equiv \emptyset$ denotes an empty type constraint system.

**Definition B.2** (Type environments)**.**

1. $\mathcal{E}_0 \equiv \emptyset$ is an empty type environment.

2. If $\mathcal{E}$ is a type environment, $N$ is a name not appearing in $\mathcal{E}$, and $\tau$ is a type, then $\mathcal{E}' \equiv \mathcal{E} \cup \{N : \tau\}$ is a type environment, and we write

$$\mathcal{E}'(N) = \tau.$$

## B.1 Type equality and matching; consistency

The following rules describe how a type constraint system produces judgements of type equality and matching. There are two base cases for each:

one which handles judgements added directly to the type constraint system (e.g. from an interface), and one which handles judgements on explicit `objecttypes` (which may use the constraint system recursively).

When we need to take apart an explicit `objecttype`, we use the notation $objectType\langle\mu, \rho\rangle$. Here $\mu$ represents the object's MyType name, which may be different than the string "`mytype`" since the implementation occasionally alpha-converts MyType names to avoid name conflicts during substitution. $\rho$ represents a record of the object's methods.

**Table B.1** (Equality and matching rules)**.**

| | |
|---|---|
| Equality: direct | $\dfrac{(\tau_1 = \tau_2 \in \mathcal{C}) \vee (\tau_2 = \tau_1 \in C)}{\mathcal{C} \vdash \tau_1 = \tau_2}$ |
| Equality: explicit | $\dfrac{\begin{array}{c}\tau_1 = objectType\ \langle\mu_1, \rho_1\rangle \\ \tau_2 = objectType\ \langle\mu_2, \rho_2\rangle \\ \mathcal{C} \vdash \rho_1 =_\alpha \rho_2[\mu_1/\mu_2]\end{array}}{\mathcal{C} \vdash \tau_1 = \tau_2}$ |
| Equality: induction | $\dfrac{\begin{array}{c}\mathcal{C} \vdash \tau_1 = \tau_2 \\ \mathcal{C} \vdash \tau_2 = \tau_3\end{array}}{\mathcal{C} \vdash \tau_1 = \tau_3}$ |
| Equality: matching | $\dfrac{\begin{array}{c}\mathcal{C} \vdash \tau_1 <\# \tau_2 \\ \mathcal{C} \vdash \tau_2 <\# \tau_1\end{array}}{\mathcal{C} \vdash \tau_1 = \tau_2}$ |
| Matching: direct | $\dfrac{\tau_1 <\# \tau_2 \in \mathcal{C}}{\mathcal{C} \vdash \tau_1 <\# \tau_2}$ |
| Matching: explicit | $\dfrac{\begin{array}{c}\tau_1 = objectType\ \langle\mu_1, \rho_1\rangle \\ \tau_2 = objectType\ \langle\mu_2, \rho_2\rangle \\ \mathcal{C} \vdash \rho_2[\mu_1/\mu_2] \subseteq_\alpha \rho_1\end{array}}{\mathcal{C} \vdash \tau_1 <\# \tau_2}$ |
| Matching: induction | $\dfrac{\begin{array}{c}\mathcal{C} \vdash \tau_1 <\# \tau_2 \\ \mathcal{C} \vdash \tau_2 <\# \tau_3\end{array}}{\mathcal{C} \vdash \tau_1 <\# \tau_3}$ |

**Definition B.3** (Consistency)**.** A type constraint system $\mathcal{C}$ is consistent if, for any pair of explicit `objecttype`s

$$\tau_1 = \mathit{objectType}\,\langle \mu_1, \rho_1 \rangle$$
$$\tau_2 = \mathit{objectType}\,\langle \mu_2, \rho_2 \rangle$$

both of the following conditions are satisfied:

1. If $\mathcal{C} \vdash \tau_1 = \tau_2$ then $\mathcal{C} \vdash \rho_1 =_\alpha \rho_2[\mu_1/\mu_2]$.
2. If $\mathcal{C} \vdash \tau_1 <\!\!\# \; \tau_2$ then $\mathcal{C} \vdash \rho_2[\mu_1/\mu_2] \subseteq_\alpha \rho_1$.

We use the notation $\rhd$ to mean "produces a consistent type constraint system". For instance, if $\mathcal{C}_1$ and $\mathcal{C}_2$ are type constraint systems, then

$$\mathcal{C}_1 \cup \mathcal{C}_2 \rhd \mathcal{C}_3$$

means "the union of $\mathcal{C}_1$ and $\mathcal{C}_2$ is a consistent type constraint system $\mathcal{C}_3$."

## B.2 Interfaces and declarations

The following rules specify how the declarations from a list of imported interfaces are turned into a type constraint system and a type environment.

**Definition B.4** (Declarations)**.**

1. If $T$ is an unqualified name (i.e. a name not including the scope resolution operator ::) and $\tau$ is either a named type or an explicit `objecttype` with no free variables, then $T = \tau$ and $T <\!\!\# \; \tau$ are declarations (specifically, they are type declarations).
2. Additionally, if $N$ is an unqualified name and $\tau$ is either a named type or an explicit `objecttype` with no free variables, then $N : \tau$ is a declaration (specifically, a constant declaration).

**Table B.2** (Declaration-list rules)**.**

| | |
|---|---|
| Empty interface | $\mathcal{C}, \mathcal{E} \vdash \emptyset \rhd \mathcal{C}, \mathcal{E}$ |
| Equality declaration | $\dfrac{\mathcal{C} \cup \{T = \tau\} \rhd \mathcal{C}' \quad \mathcal{C}', \mathcal{E} \vdash \mathit{otherDecls} \rhd \mathcal{C}'', \mathcal{E}'}{\mathcal{C}, \mathcal{E} \vdash T = \tau; \mathit{otherDecls} \rhd \mathcal{C}'', \mathcal{E}'}$ |

Matching declaration

$$\frac{\mathcal{C} \cup \{T <\!\!\# \tau\} \rhd \mathcal{C}' \quad \mathcal{C}', \mathcal{E} \vdash \mathit{otherDecls} \rhd \mathcal{C}'', \mathcal{E}'}{\mathcal{C}, \mathcal{E} \vdash T <\!\!\# \tau; \mathit{otherDecls} \rhd \mathcal{C}'', \mathcal{E}'}$$

Constant declaration

$$\frac{\mathcal{C} \cup \{N : \tau\} \rhd \mathcal{C}' \quad \mathcal{C}', \mathcal{E} \vdash \mathit{otherDecls} \rhd \mathcal{C}'', \mathcal{E}'}{\mathcal{C}, \mathcal{E} \vdash N : \tau; \mathit{otherDecls} \rhd \mathcal{C}', \mathcal{E}''}$$

**Definition B.5** (Fully-qualified names)**.** A fully-qualified name consists of an interface name $M$ and a declaration name $N$, and is denoted $M{::}N$.

The function QUALIFY takes an interface name and a list of declarations and affixes the interface name to the name of each declaration, producing a new declaration list in which each declaration name is fully-qualified.

**Definition B.6** (Interface systems)**.**

1. Following the *interfaceUnit* rule defined in Table A.1, an interface consists of a name $M$, a list of imported interface names $\mathit{imps} \equiv \{\mathit{imp}_1, \ldots, \mathit{imp}_n\}$, and a list of declarations $\mathit{decls} \equiv \{\mathit{decl}_1, \ldots, \mathit{decl}_m\}$. An interface will be denoted by $\mathit{interfaceUnit}\,\langle M, \mathit{imps}, \mathit{decls}\rangle$.
2. $\mathcal{I}_0 \equiv \emptyset$ is an empty interface system.
3. If $\mathcal{I}$ is an interface system and $\mathit{interfaceUnit}\,\langle M, \mathit{imps}, \mathit{decls}\rangle$ is an interface, then $\mathcal{I}' \equiv \mathcal{I} \cup \{\mathit{interfaceUnit}\,\langle M, \mathit{imps}, \mathit{decls}\rangle\}$ is an interface system, and we write
$$\mathcal{I}'(M) = \langle \mathit{imps}, \mathit{decls}\rangle\,.$$

**Table B.3** (Import-list rules)**.**

Empty import list

$$\mathcal{I}, \mathcal{C}, \mathcal{E} \vdash \emptyset \rhd \mathcal{C}, \mathcal{E}$$

Import list

$$\frac{\mathcal{I}(\mathit{imp}_1) = \langle \mathit{imps}, \mathit{decls}\rangle \quad \mathcal{I}, \mathcal{C}, \mathcal{E} \vdash \mathit{imps} \rhd \mathcal{C}', \mathcal{E}' \quad \mathcal{C}, \mathcal{E} \vdash \textsc{qualify}(\mathit{decls}) \rhd \mathcal{C}'', \mathcal{E}'' \quad \mathcal{I}, \mathcal{C}'', \mathcal{E}'' \vdash \mathit{otherImports} \rhd \mathcal{C}''', \mathcal{E}'''}{\mathcal{I}, \mathcal{C}, \mathcal{E} \vdash \mathit{imp}_1; \mathit{otherImports} \rhd \mathcal{C}''', \mathcal{E}'''}$$

This last rule simply says that when an interface is imported, first all the interfaces *it* imports are recursively imported, and then the interface's own declarations are added (in fully-qualified form) to the current constraint graph and type environment.

## B.3 Compilation units

The following rules specify how compilation units (which are either programs, interfaces, or modules) are type-checked. The only information carried between compilation units is the interface system $\mathcal{I}$, which represents the interfaces that have already been processed and are therefore available for import. Each invocation of the type-checker starts with a fresh interface system, which is empty except for standard-library interfaces.

**Table B.4** (Compilation unit rules)**.**

Base case $$\mathcal{I} \vdash \emptyset \hookrightarrow \mathcal{I}$$

Program $$\frac{\begin{array}{c}\mathcal{I}, \mathcal{C}_0, \mathcal{E}_0 \vdash imps \rhd \mathcal{C}_{imp}, \mathcal{E} \\ abbrevs \rhd \mathcal{C}_{abbrev} \\ \mathcal{C}_{abbrev} \cup \mathcal{C}_{imp} \rhd \mathcal{C} \\ \mathcal{C}, \mathcal{E} \vdash block \\ \mathcal{I} \vdash otherUnits \hookrightarrow \mathcal{I}'\end{array}}{\mathcal{I} \vdash programUnit\ \langle N, imps, abbrevs, block\rangle\ ;\ otherUnits \hookrightarrow \mathcal{I}'}$$

This says that to type-check a program, we first gather its imports into a constraint system $\mathcal{C}_{imp}$ and type environment $\mathcal{E}$. We also gather constraints based on the program's own type abbreviations ($\mathcal{C}_{abbrev}$) and put them all into one constraint system $\mathcal{C}$, which we ensure is consistent and then use to type-check the program body. (The rules for judging $\mathcal{C}, \mathcal{E} \vdash block$ may be found, with the rest of the core language type-checking rules, in [Pet96].)

Interface $$\frac{\begin{array}{c}\mathcal{I}, \mathcal{C}_0, \mathcal{E}_0 \vdash imps \rhd \mathcal{C}_{imp}, \mathcal{E} \\ \mathcal{C}_{imp}, \mathcal{E} \vdash decls \rhd \mathcal{C}, \mathcal{E}' \\ \mathcal{I} \cup \{\langle N, imps, decls\rangle\} \vdash otherUnits \hookrightarrow \mathcal{I}'\end{array}}{\mathcal{I} \vdash interfaceUnit\ \langle N, imps, decls\rangle\ ;\ otherUnits \hookrightarrow \mathcal{I}'}$$

To type-check an interface, we gather its imports into a constraint system and environment (just as we do for programs) and then incorporate the contraints from the interface itself (*decls*). If the resulting constraint system is consistent, we add the interface to the interface system.

$$\text{Implementation}\ \frac{\begin{array}{c}\mathcal{I}, \mathcal{C}_0, \mathcal{E}_0 \vdash (names \cup imps) \rhd \mathcal{C}_{imp}, \mathcal{E} \\ abbrevs \rhd \mathcal{C}_{abbrev} \\ \bigcup_{n \in names} \bigcup_{d \in \mathcal{I}(n).decls} \{d.name = n{::}d.name\} \rhd \mathcal{C}_Q \\ \mathcal{C}_{abbrev} \cup \mathcal{C}_{imp} \cup \mathcal{C}_Q \rhd \mathcal{C} \\ \forall n \in names\ \forall d \in \mathcal{I}(n).decls\ \textsc{has-def}(d, \mathcal{C}, abbrevs, consts) \\ \mathcal{C}, \mathcal{E} \vdash consts \\ \mathcal{I} \vdash otherUnits \hookrightarrow \mathcal{I}'\end{array}}{\begin{array}{c}\mathcal{I} \vdash\ implementationUnit\ \langle names, imps, abbrevs, consts\rangle\ ; \\ otherUnits \hookrightarrow \mathcal{I}'\end{array}}$$

$$\text{where } \textsc{has-def}(T = \tau, \mathcal{C}, abbrevs, consts) = \exists \sigma\ \begin{cases}\sigma = objectType\ \langle \mu, \rho\rangle \\ \mathcal{C} \vdash T = \sigma\end{cases} \vee \exists a \in abbrevs\ T = a.name$$

$$\textsc{has-def}(T <\!\!\# \tau, \mathcal{C}, abbrevs, consts) = \exists \sigma\ \begin{cases}\sigma = objectType\ \langle \mu, \rho\rangle \\ \mathcal{C} \vdash T = \sigma\end{cases} \vee \exists a \in abbrevs\ T = a.name$$

$$\textsc{has-def}(N : \tau, \mathcal{C}, abbrevs, consts) = \exists c \in consts\ N = c.name$$

Type-checking an implementation begins by gathering constraints from the imports and type abbreviations, just as in the rule for programs. We also include the interfaces being implemented by this module (*names*). We then create a type constraint system $\mathcal{C}_Q$ that contains a constraint for each declaration in each of the interfaces being implemented, equating the declarations' short names to their fully-qualified names. Since types are referred to by short names in the module body but fully-qualified names in the interfaces, explicitly linking the two is necessary. Then we ensure that each type and constant declared in one of the implemented interfaces is fully defined (using the predicate HAS-DEF). Finally, we type-check the module body and recursively type-check any remaining compilation units.

HAS-DEF works as follows. For type declarations ($T = \tau$ or $T <\!\!\# \tau$) it first checks to see if the type is already fully defined by the constraint system $\mathcal{C}$. That is, it looks for a type $\sigma$ that is an explicit `objecttype` and has $\mathcal{C} \vdash T = \sigma$. If this is not the case, $T$ is required to be defined in the module implementation's list of type abbreviations (*abbrevs*). For constant declarations ($N : \tau$), HAS-DEF checks that $N$ is defined in the module implementation's list of constants (*consts*).

# Appendix C

# Example programs

In this appendix, we give a pair of complete example *LOOM* programs using the new module system.

## C.1 Extending a closed-source module

This example expands upon the GUI widget example discussed in §3.5.1. First, a widget module is defined with two interfaces, a stingy one and a detailed one. Then a module extending the widget is defined using the detailed interface. Finally, a program using both kinds of widgets interchangeably is demonstrated.

```
-- GUI widget extension test for LOOM
-- Written by Nathan Reed, spring 2008

--------------------------------------------------
-- A basic GUI widget module with two interfaces. --
--------------------------------------------------

interface Widget;
  -- Partially reveal an object type for the widget
  Widget <# objecttype
    redraw: proc ();
    setPos: proc (integer, integer);
  end;

  -- A factory function to let clients create widgets
  makeWidget: func () : Widget;

  -- A function that draws two widgets at once.
```

```
  -- This takes #Widget as a parameter so that any
  -- customized extension of Widget can be used.
  drawWidgets: proc (#Widget, #Widget);
end;


interface WidgetFull;
  import Widget;

  -- Fully revealing the widget's object type
  Widget = Widget::Widget;
  Widget = objecttype
    -- The same methods we saw before
    redraw: proc ();
    setPos: proc (integer, integer);

    -- Also reveal some internal methods
    drawBorder:   proc ();
    drawContents: proc ();
  end;

  -- A class generating Widgets, which clients
  -- can subclass
  WidgetClass: classtype
    var posX, posY : integer;
    methods
      redraw: proc ();
      setPos: proc (integer, integer);
      drawBorder: proc ();
      drawContents: proc ();
  end;
end;


module implements Widget, WidgetFull;
  import IO;

  -- Note: no need to give definition for Widget type since it is
  -- fully revealed by the WidgetFull interface.

  const
    class WidgetClass
      var
        posX = 0, posY = 0 : integer;
      methods
```

```
        procedure redraw ()
        begin
          IO::printString("Redrawing Widget\n");
          drawBorder();
          drawContents();
        end;

        procedure setPos (x, y : integer)
        begin
          posX := x;
          posY := y;
        end;

        procedure drawBorder ()
        begin
          IO::printString("...drawing border\n");
        end;

        procedure drawContents ()
        begin
          IO::printString("...drawing contents\n");
        end;
    end;

    function makeWidget () : Widget
    begin
      return new WidgetClass;
    end;

    procedure drawWidgets (widget1, widget2 : #Widget)
    begin
      widget1.redraw();
      widget2.redraw();
    end;

end;


-----------------------------------------------------
-- An extension module defining a custom widget. --
-----------------------------------------------------

interface CustomWidget;
  import Widget;
```

```
  -- Declare our own custom widget type to match
  -- the original widget type
  CustomWidget <# Widget::Widget;

  -- Reveal an additional method of our custom widget
  CustomWidget <# objecttype
    setCustomState: proc (string);
  end;

  -- Factory function
  makeCustomWidget: func () : CustomWidget;
end;


module implements CustomWidget;
  import IO, WidgetFull;

  type
    CustomWidget = objecttype
      redraw: proc ();
      setPos: proc (integer, integer);
      drawBorder:   proc ();
      drawContents: proc ();
      setCustomState: proc (string);
    end;

  const
    class CustomWidgetClass
      inherit WidgetFull::WidgetClass modifying redraw;
      var
        customState = "" : string;
      methods
        procedure redraw ()
        begin
          IO::printString("Redrawing CustomWidget\n");
          IO::printString("...custom state is: " ^ customState ^ "\n");
          drawContents();
        end;

        procedure setCustomState (newState : string)
        begin
          customState := newState;
        end;
    end;
```

```
    function makeCustomWidget () : CustomWidget
    begin
      return new CustomWidgetClass;
    end;

end;


-------------------------------------------
-- An program using both kinds of widgets. --
-------------------------------------------

program WidgetTest;
  import IO, Widget, CustomWidget;

var
  myWidget : Widget::Widget;
  myCustomWidget : CustomWidget::CustomWidget;

begin
  -- Create some widgets
  myWidget := Widget::makeWidget();
  myCustomWidget := CustomWidget::makeCustomWidget();

  -- Set some state
  myWidget.setPos(12, 34);
  myCustomWidget.setPos(56, 78);
  myCustomWidget.setCustomState("Hello, world!");

  -- Treat the custom widget as an ordinary widget
  Widget::drawWidgets(myWidget, myCustomWidget);
end;
```

## C.2 Providing distinct functionality to different clients

This example expands upon the student information system example discussed in §3.5.2. First, an `InfoSys` interface is defined revealing very basic information about a `Student` type, then two more generous interfaces (`Registrar` and `FinancialAid`) are defined that reveal more information about `Students`. Then, a program importing both of these interfaces is demonstrated.

```
-- Student information system test for LOOM
-- Written by Nathan Reed, spring 2008

-----------------------------------------------------------------
-- A student information system module with three interfaces. --
-----------------------------------------------------------------

interface InfoSys;
  -- Declare a student type, revealing only the
  -- information everyone can see
  Student <# objecttype
    getName: func () : string;
    getAddress: func () : string;
  end;

  -- Look up a student by ID number
  getStudent: func (integer) : Student;
end;


interface Registrar;
  import InfoSys;

  -- Reveal additional registrar-only information about Student
  Student = InfoSys::Student;
  Student <# objecttype
    getGPA: func () : real;
    getNumCredits: func () : real;
  end;
end;


interface FinancialAid;
  import InfoSys;

  -- Reveal additional financial-aid-only information about Student
  Student = InfoSys::Student;
  Student <# objecttype
    getIncome: func () : real;
    getLoansTotal: func () : real;
  end;
end;


module implements InfoSys, Registrar, FinancialAid;
```

```
type
  Student = objecttype
    -- The methods revealed in the interfaces
    getName: func () : string;
    getAddress: func () : string;
    getGPA: func () : real;
    getNumCredits: func () : real;
    getIncome: func () : real;
    getLoansTotal: func () : real;

    -- Additional methods
    populateFromDatabase: proc (integer);
  end;

const
  class StudentClass
    var
      name, address : string;
      GPA, numCredits : real;
      income, loansTotal : real;
    methods
      function getName () : string
      begin
        return name;
      end;
      function getAddress () : string
      begin
        return address;
      end;
      function getGPA () : real
      begin
        return GPA;
      end;
      function getNumCredits () : real
      begin
        return numCredits;
      end;
      function getIncome () : real
      begin
        return income;
      end;
      function getLoansTotal () : real
      begin
        return loansTotal;
```

```
        end;

        procedure populateFromDatabase (studentId : integer)
        begin
          -- Since we don't actually have a database,
          -- just put in some made-up data.
          name := "John Smith";
          address := "123 Ocean Avenue\nSanta Monica, CA";
          GPA := 3.86;
          numCredits := 102.4;
          income := 65536.0;
          loansTotal := 16384.0;
        end;
    end;

    function getStudent (studentId : integer) : Student
    var s : Student;
    begin
      s := new StudentClass;
      s.populateFromDatabase(studentId);
      return s;
    end;

end;


-----------------------------------------------------------------
-- A program using both Registrar and FinancialAid interfaces. --
-----------------------------------------------------------------

program StudentTest;

-- By importing all three interfaces, we should have access
-- to the union of the Student methods from each.
import InfoSys, Registrar, FinancialAid, IO;

var
  theStudent: InfoSys::Student;
  GPA, income: real;

begin
  -- Look up some student
  theStudent := InfoSys::getStudent(47);
  IO::printString("Got a student named "
                  ^ theStudent.getName() ^ "\n");
```

```
  -- We can get the student's GPA
  GPA := theStudent.getGPA();
  IO::printString("Student's GPA is ");
  IO::printReal(GPA);
  IO::printString("\n");

  -- We can get the student's income
  income := theStudent.getIncome();
  IO::printString("Student's income is ");
  IO::printReal(income);
  IO::printString("\n");
end;
```