A Higher-order Module Discipline with Separate Compilation, Dynamic Linking, and Pickling

DRAFT

Denys Duchier, Leif Kornstaedt, Christian Schulte, Gert Smolka

Programming Systems Lab
DFKI and Universität des Saarlandes
Postfach 15 11 50, D-66041 Saarbrücken, Germany
{duchier,kornstae,schulte,smolka}@ps.uni-sb.de

September 29, 1998

Abstract

We present a higher-order module discipline with separate compilation and concurrent dynamic linking. Based on first-order modules one can program security policies for systems that link modules from untrusted locations (e.g., Java). We introduce a pickling operation that writes persistent clones of volatile, possibly higher-order data structures on the file system. Our pickling operation respects lexical binding. Our module discipline is based on functors, which are annotated functions that are applied to modules and return modules. Pickled computed functors can be used interchangeably with compiled functors. In contrast to compiled functors, pickled computed functors can carry computed data structures with them, which has significant practical applications.

1 Introduction

Modularization is an essential technique for developing and maintaining software systems [12, 16]. Programming languages support modularization by replacing complete programs with program fragments called *module definitions*. Module definitions reside in individual files and are *compiled separately*. Execution of a module definition installs a *module*. Module definitions typically *import* modules,

which are identified by file names. *Dynamic linking* [4] is a scheme, where execution starts with a single compiled module definition and further module definitions are loaded and installed only once they are needed. The same module definition can be used by many different applications.

Modula-2 [16], Modula-3 [10], Oberon [13], Objective Caml [8] and Java [2] are examples of languages that support modules and separate compilation. Oberon and Java also support dynamic linking.

A module definition specifies a function that takes modules as arguments and returns a module. This observation relates module definitions to definitions of functional procedures [3]. We will use the term *functor* to stand for the annotated function specified by a module definition. The annotations specify for each imported module an external name and a type. Our functors are different from SML's [9] functors in that SML's functors do not have annotations.

In functional languages [1, 9], procedures are accommodated as first-class citizens, which provides for powerful programming techniques. Similarly, we will accommodate functors and modules as first-class citizens. This means that program variables can designate modules and functors, and that functor definitions can be nested into functor definitions.

We will present two significant practical applications of first-class modules and first-class functors.

First-class modules make it possible to program flexible security policies for systems that obtain functors from possibly untrusted locations in the Internet (e.g., Java). A security policy may, for instance, completely disallow operations on the local file system or query the user if such an operation is invoked.

For our second application we introduce pickling and unpickling operations. Pickling writes a persistent clone of a volatile data structure on the file system, called a *pickle*. Unpickling reads a persistent clone from the file system and obtains a volatile clone. With first-class functors, pickling offers the possibility to write computed functors on the file system. Pickled computed functors can be used in the same way as compiled module definitions.[1] In contrast to compiled functors, computed functors can carry computed data structures with them. This matters since

1. a computed data structure can now be loaded together with a functor rather than being computed a new for each process using it.

2. the functors needed to compute a data structure are not needed by the processes using the data structure.

[1] This comes for free if the compiler is implemented in the same language it compiles. Then the compiler simply executes the compiled module definition and pickles the obtained functor.

To the best of our knowledge, these two application of first-class modules and functors have not been proposed before. The same holds for our general pickling operation, which respects lexical scoping and accommodates first-class procedures and functors. Modula-3 [10] has an implementation-dependent pickling operation that accommodates first-order data structures but breaks security invariants for procedures.

Higher-order module disciplines have been investigated thoroughly in the context of ML (e.g., [9, 15, 6, 7]). This work focuses on expressive static type systems and mostly ignores separate compilation. In contrast to our approach, functor definitions are not seen as compilation units. Compilation units appear as a separate notion in Objective Caml [8].

Our module discipline assumes a dynamically typed base language. Reconciling our module discipline with a static type system is a difficult research issue. However, it is important to have a simple reference model of a higher-order module discipline and explore its practical applications. This provides practical guidance for research on the technical difficult issues of static typing. As it comes to typing, SML and Java provide two important and radically different reference points. In contrast to SML, Java mixes static with dynamic type checking. Dynamic typing is a must for dynamically linking systems that load functors from untrusted locations.

Our module system was designed and implemented for the next version of Oz [14, 11], a dynamically typed, concurrent and higher-order language that bases synchronization of threads on logic variables and provides powerful primitives for constraint programming. The use of pickled computed functors in addition to compiled functors has proven essential for the applications we have adapted to the new module discipline.

Our presentation of the module discipline employs a dynamically typed variant of SML's base language extended with concurrent threads. This is a rather generic choice that does not really commit us to a particular programming language. To provide for a sufficiently rigorous explanation of dynamic linking, we use futures as in Multilisp [5]. To our knowledge, this is the first rigorous model of lazy and concurrent linking (which is employed in Java).

The paper is organized as follows. Section 2 starts with a first-order module discipline with eager linking. Section 3 introduces module types and link-time type checking. Section 4 adds concurrent lazy linking based on module managers. Section 5 adds first-class modules and shows how they can be used to program security policies. Section 6 adds pickles. Section 7 adds first-class functors and discusses pickled computed functors.

2 Basic Notions

We start with a simple module discipline where all linking is done before the actual execution starts. We assume a sequential base language. Functors and modules are not first-class.

We assume that the language is implemented with a *compiler* and a *virtual machine* (VM). The compiler translates *functor definitions* into *functor files*. The VM loads functor files, links the obtained functors and executes them. Functor definitions and functor files reside in a file system whose file names are strings called *URLs*. The file system should be thought of as a combination of the local file system and the Internet-based virtual file system provided by the URL infrastructure. The compiler and the VM are invoked at the level of the operating system.

Modules are volatile structures that exist in a virtual machine. They take the form of records whose *fields* are often higher-order (e.g., procedures). We distinguish between *system modules* and *application modules*. System modules provide access to system resources like the file system and the window system. System modules are provided by the VM. Application modules are computed by functors.

Every module is associated with a *module name*. There are different names for system modules and application modules. The name of an application module is the URL from which the functor that computed it was obtained.

A *functor* is an annotated procedure. The procedure is applied to modules and returns a module. The annotation fix a name for every argument module, possibly relative to the URL from which the functor was obtained.

A *functor definition* is a text residing on a file. Functor definitions are compiled with respect to a *base environment*, that provides the basic operations of the language (e.g., operations for arithmetic). We assume that the basic operations of the language do not use system resources like input and output channels. Access to system resources must be obtained through imported system modules.

The VM is started with a URL, called *root URL*. From the root URL the VM obtains the so-called *root functor*. The root functor specifies names for its argument modules, possibly relative to the URL from which it was obtained. If the root functor needs further application modules, the VM continues by loading the respective functors. The VM continues loading functors until it has built the complete *dependency graph*.

The nodes of the dependency graph are module names. The links of the dependency graph point from a module name *N* to the names of the modules that *N imports*. Every node of the dependency graph is reachable from the root URL. Names of system modules appear as leaves of the dependency graph. The dependency graph must be acyclic.

Once all functors of the dependency graph are loaded, the VM applies the

functors in bottom-up order. It terminates when the application of the root functor terminates. The choice of the precise bottom-up order (e.g., from left to right) in which functors are applied is left to the VM.

The reader may wonder why the VM can provide actual services if all it does is applying functors. The answer is that a functor may use the input/output facilities of an imported module (e.g., a system module). Typically, this will be done by the root functor, which will only return a trivial module.

The VM can be called with the root URL and further command line arguments. The command line arguments can be made visible to the application functors through a system module.

The following things can go wrong while the VM is running.

1. The VM may not be able to obtain a functor from a URL.
2. The functors at the URLs may define a cyclic or an infinite dependency graph.
3. A functor application may return an (error) exception. (For instance, because a field of an imported module does not exist or has the wrong type).

If something goes wrong, the VM prints an error message and terminates.

Note that in the current model functors are applied exactly once. This means that they cannot be used to obtain multiple instances of generic modules. In Section 5, we will extend our model so that is provides for multiple applications of functors.

Any dynamically typed language with lexically scoped first-class procedures (e.g, Scheme) can be rearranged to support the described module discipline.

3 Module Types

Even in a dynamically typed language we can have some type checking at the module level. To do so, we need *module types*. A functor now states a type for each argument module and the result module. Moreover, we need a relation that defines whether a module type T_1 *supports* a module type T_2. If a functor requires a type T for an argument module, any module that has a type that supports T can be used as argument.

The most primitive module types would just list which field names are expected or provided. Then T_1 supports T_2 if T_1 lists at least the field names listed by T_2.

Type checking is needed at compile time and at link time. Type checking at link time ensures that the support relation is satisfied for each link of the dependency graph. Type checking at compile time may assume that a functor definition

declares the imported and exported field names and checks that only declared import fields are used and all declared export fields are provided. With suitable syntax the compiler may alternatively infer all used import fields and all provided export fields.

One may use types for fields like integer, procedure, or record. In this case the compiler must check that declared field types are consistently used.

In a statically typed language, one can declare and check more expressive field types. Now it becomes to inconvenient to declare in each functor definition the types of all imported and exported fields.

In Java and Oberon, one declares just the types of the export fields. The types of the import fields are known by the compiler if they belong to system modules and are otherwise obtained by the compiler from the imported functor files.

Modula-2 and Objective Caml use functor definitions and corresponding interface definitions. An interface definition states exported fields together with their types. A functor definition can only be compiled if the compiled interface definitions for the functor and its imported functors are available. Implementations of Modula-2 and Objective Caml typically represent module types as finger prints (e.g., MD5 check sums) and check equality of finger prints at link time.

4 Concurrent Linking

We now move to a lazy linking strategy where functors are applied in a top-down order with respect to the dependency graph. Moreover, a module is only installed when one of its fields is accessed. This means that execution of functors and installation of modules are interleaved. This by-need strategy is reminiscent of lazy functional programming. It is different in that computation is eager except that the installation of modules is delayed.

We also move to a base language with concurrent threads. In a concurrent setting lazy linking is particularly useful since threads that do not block on the installation of a module can continue their computation.

To explain lazy linking, we use the notion of futures from Multilisp [5]. A *future* is a placeholder for a data structure that is not yet computed. Once the data structure is computed, it replaces the future such that the future disappears. Threads can block on the event that a future is needed. This way the data structure that will replace a future can be computed by a thread that waits until the future is needed.

We assume that the VM hosts a *module manager* that is responsible for linking and installation of modules. The module manager host a *module table* that maps module names to futures or pairs (M,T) consisting of a module M and its type T. A module name that is mapped to a future is linked but not yet installed, and a

module name that is mapped to a pair (M, T) is linked and installed. When the VM starts, all system modules are linked.

The module manager has a method

$\mathsf{link}(U, T)$

that takes a module name U and a module type T and returns a future M_f representing the module identified by U. When a field of M_f is accessed, the following happens:

1. If the module identified by U is not yet installed, it is installed.
2. If the type of the installed module supports T, the future M_f is replaced with the installed module.

Concurrent applications of the link method are served under mutual exclusion. The VM starts with linking and requesting the root module. The rest happens through threads spawned by the link method.

The method $\mathsf{link}(U, T)$ is defined as follows:

1. If U is not in the domain of the module table, then:
   (a) Create a new future E_f and extend the module table such that it maps U to E_f.
   (b) Spawn a new thread as follows (*installation of* U):
      i. Wait until E_f is needed.
      ii. Load a functor F from U; raise a linking error if this fails.
      iii. Link the argument modules of F.
      iv. Apply F to the argument modules (some of them are possibly represented as futures) and obtain a module M.
      v. Replace E_f with the pair (M, T_F), where T_F is the result type of F.
2. Let E be the entry to which the module table maps U.
3. Create a new future M_f.
4. Spawn a new thread as follows (*type checking for* U):
   (a) Wait until M_f is needed.
   (b) Wait until E is a pair (M, T_M). (If E is a future, this will request its elimination.)
   (c) If T_M supports T, then replace M_f with M, else raise a linking type error.

5. Return M_f.

Note that concurrent linking makes its possible to have *cyclic dependency graphs*. Java admits cyclic dependency graphs since it allows for mutually dependent classes.

In the sequential discipline, the VM terminates once the application of the root functor terminates. In a concurrent setting it is preferable to have an explicit shutdown operation since, for instance, threads spawn by the root functor may still be active when its application terminates. The shutdown operation should be made available through a system module.

5 First-class Modules

We now move to a setting where modules are first-class citizens. In a dynamically typed language this is a straightforward extension since our discipline represents modules anyway as records.

We assume that functors provide first-class access to their argument modules.

First-class modules become interesting if we provide the possibility to freely create new module managers. Let us assume that a system module provides an operation

```
link : moduleTable * URL * moduleType -> module
```

where `moduleTable` stands for lists of triples

```
moduleName * module * moduleType
```

An application

```
link(mtab,u,t)
```

will create a new module manager whose initial module table is specified by `mtab`. Then the link method of the new module manager will be applied to `(u,t)` and deliver the result of the link operation.

With the link operation we can apply a functor more than once. Hence we can now have generic modules.

Another application of the link operation is the implementation of security policies. Suppose we want to install a functor from an untrusted URL. We can do this with a new module manager whose initial module table provides only secured variants of the system modules. The secured variants can for instance query the user if the the untrusted application wants to access the local file system. We have

sufficient expressivity for constructing secured variants of system modules since modules are records and the language is higher-order.

Java's class loaders provide part of the expressivity of our link operation. Java looses expressivity since classes are in general not first-class.

6 Pickles

We now extend our model with two operations pickle and unpickle. The *pickle operation* obtains a portable description of a data structure, called *a pickle*, in a VM and writes it on a file. The *unpickle operation* reads a pickle from a file and creates a clone of the original data structure in the VM in which the unpickle operation is invoked. In other words, pickling creates a persistent clone of a volatile data structure, and unpickling creates a volatile clone of a persistent clone.

We are interested in pickles since we will move to a setting where functors are first-class citizens and can hence be pickled. Things will be generalized such that module managers can load pickled functors produced by VMs as well as functor files produced by the compiler.

We need to make precise what we understand under a data structure and which data structures can be pickled. To do so, we assume that our language has the data structures of SML.

We assume an execution model that separates control structures (i.e., threads) from data structures. All data structures reside in an abstract store, whose states take the form of a directed graph. The nodes of the graph represent entities like integers, records, variants (obtained by constructors), assignable references cells, procedures (i.e., closures), and operations (i.e., built-in procedures). Procedures are nodes whose departing links point to lexically bound nodes. For instance, the SML expression

```
let val x = fn y => z(y) in ... end
```

will bind x to a procedure node with one departing link that points to the node bound to z.

Given an execution state, the data structure associated with a node x in the abstract store is the maximal subgraph of the abstract store that is reachable from x.

We distinguish between two types of operations (i.e., built-in procedures). *Global operations* have the same semantics in every VM (e.g., addition of numbers or creation of threads). *Local operations* affect the resources of a particular VM and are hence tied to a particular VM. All operations that are available through the base environment at compile time are global. Access to local operations can

only be obtained through system modules, which are tied to the VM in which they exist.

A data structure in a VM can be pickled if and only if it does not contain futures or local operations. By excluding futures, we make sure that they are not cut off from the threads that are supposed to eliminate them. By excluding local operations, we ensure that loading a pickle will not create a data structure that captures local operations of the loading VM. This is an essential security property of our model (see also Section 5).

If the pickle operation encounters a future, it requests it and blocks. Once the future is eliminated it resumes. This way pickling is compatible with lazy linking. If the pickle operation encounters a local operation, it raises an error exception.

Next we make precise what we mean by a clone of a data structure. Let *x* be the original data structure and *y* be a clone of *x*. Then the graphs reachable from *x* and *y* must be graph isomorphic with respect to their roots *x* and *y*. Moreover, the reference cells reachable from *x* must all be different from the reference cells reachable from *y*. Finally, if we redirect all external links into the graph rooted by *x* to the respective nodes of the graph rooted by *y*, no difference must be observable when the VM proceeds.

Modula-3 [10] offers pickles that depend on the implementation and that are different from ours as it comes to procedures and operations. Pickles in Modula-3 do not contain procedures. Instead of the actual procedures their symbolic names will be pickled. When the VM unpickles a pickle, it will try to resolve the symbolic names with the procedures it knows. Operations are treated like procedures.

7 Computed Functors

We now move to a language with first-class functors. This means that functor definitions can contain nested functor definitions, and that functors computed by nested definitions can be referred to through program variables.

We distinguish between *compiled functors* and *computed functors*. A compiled functor is obtained by compilation of a functor definition. Computed functors are obtained by executing compiled functors whose definitions contain nested functor definitions. Compiled functors can only have lexical bindings to the data structures of the base environment (see Section 2). Computed functors can have lexical bindings to all data structures that the creating compiled functors supply to their definitions.

Since computed functors are first-class, we can pickle them. This becomes useful, if we assume that module managers can load compiled functors as well as pickled computed functors. Pickled computed functors can carry computed data

structures with them. This matters since

1. a computed data structure can now be loaded together with a functor rather than being computed a new for each virtual machine using it.

2. the functors needed to compute the carried with data structure are not needed by the virtual machine using it.

To make the introduction of pickled computed functors convenient and to obtain well-structured source files, the compiler should accept sugared definitions looking as follows:

computed functor
requires *Ms*
local *Ds*
functorDefinition
end

Here *functorDefinition* is an ordinary functor definition, which will create the computed functor. The compiler first transforms the sugared definition into the ordinary definition

functor
import pickle *Ms*
body
  Ds
  pickle.save(*functorDefinition*, *fileName*)
end

where pickle is the name of the system module providing pickling and *fileName* is the name of the file from which the sugared definition was obtained (modulo suitable suffixes). The compiler then compiles the obtained ordinary definition and starts the VM with the obtained functor file. The VM will execute the preparatory definitions *Ds* using the modules *Ms* and then create the computed functor and pickle it to the right file.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, 2nd edition, 1996.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, New-York, 1996.

[3] R. M. Burstall. Programming with modules as typed functional programming. In *Proc. International Conference on 5th Generation Computing Systems*, Tokio, 1984.

[4] M. Franz. Dynamic linking of software components. *Computer*, 30(3):74–81, 1997.

[5] R. H. Halstaedt. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[6] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st Symposium on Principles of Programming Languages*, pages 123–137. ACM, 1994.

[7] X. Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st Symposium on Principles of Programming Languages*, pages 109–122. ACM, 1994.

[8] X. Leroy. *The Objective Caml System, Release 1.05*. INRIA, Roquencourt, France, 1997. http://pauillac.inria.fr/caml.

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

[10] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[11] Oz. The Oz Programming System, 1997. Programming Systems Lab, Universität des Saarlandes: http://www.ps.uni-sb.de/www/oz2/.

[12] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1997.

[13] M. Reiser and N. Wirth. *Programming in Oberon: Steps beyond Pascal and Modula*. Addison-Wesley, New York, 1992.

[14] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, 1995.

[15] M. Tofte. Principal signatures for higher-order program modules. In *19th Symposium on Principles of Programming Languages*, pages 189–199. ACM, 1992.

[16] N. Wirth. *Programming in Modula-2*. Springer, Berlin, Germany, 3rd, corrected edition, 1985.