Module systems

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk
Module systems basics

“A module is a function which produces environments of a particular signature when applied to argument instances of specified signatures.”

David MacQueen
Ascribing signatures to structures (IntSet : SET) involves subtyping, including
  abstraction (turning concrete types into abstract types)
  instantiation (turning polymorphic types into concrete types)
as well as width and depth subtyping (dropping and subtyping entries).
Ascribing signatures to structures (IntSet : SET) involves subtyping, including abstraction (turning concrete types into abstract types) instantiation (turning polymorphic types into concrete types) as well as width and depth subtyping (dropping and subtyping entries).
Ascribing signatures to structures (IntSet : SET) involves subtyping, including abstraction (turning concrete types into abstract types) and instantiation (turning polymorphic types into concrete types) as well as width and depth subtyping (dropping and subtyping entries).
**Functors**: functions from modules to modules.
Abstract (and less abstract) types

-a type for MakeSet-

module MakeSet (Elem: ORDERED) :
  SET with type elem = Elem.t

-expanded signature-

SET with type elem = Elem.t
⇝ sig
  type elem = Elem.t
  type t
  val mem : Elem.t → t → bool
  ...
end

In the type of mem: t is abstract, Elem.t is shared, bool is concrete.
Using **higher-order modules** can lead to loss of type equalities:

```
module Apply (MakeSet : functor (Elem:ORDERED) → SET) (Elem : ORDERED) = MakeSet(Elem)

module IS1 = Apply(MakeSet)(Int) (* IS1.t /= Int.t *)
module IS2 = MakeSet(Int) (* IS2.t == Int.t *)
```

Leroy’s solution: extend the **path notation** to include applications

```
type t = MakeSet(Int).t
```
Module systems history

“In the case of constructions, we obtain the notion of a very high-level functional programming language, with complex polymorphism well-suited for module specification.”

The Calculus of Constructions (1988)
Thierry Coquand and Gérard Huet
<table>
<thead>
<tr>
<th>Year</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1974</td>
<td>Towards a theory of type structure</td>
<td>Reynolds</td>
</tr>
<tr>
<td>1985</td>
<td>Abstract types have existential type</td>
<td>Mitchell &amp; Plotkin</td>
</tr>
<tr>
<td>1985</td>
<td>Using dependent types to express modular structure</td>
<td>MacQueen</td>
</tr>
<tr>
<td>1988</td>
<td>The Calculus of Constructions</td>
<td>Coquand &amp; Huet</td>
</tr>
<tr>
<td>1990</td>
<td>Higher-order modules and the phase distinction</td>
<td>Harper, Mitchell &amp; Moggi</td>
</tr>
<tr>
<td>1994</td>
<td>A type-theoretic approach to higher-order modules with sharing</td>
<td>Harper &amp; Lillibridge</td>
</tr>
<tr>
<td>2010</td>
<td>F-ing modules</td>
<td>Rossberg, Russo &amp; Dreyer</td>
</tr>
</tbody>
</table>
Reading
informal notion of “abstract type” to the existential types of System F. In F, values
Mitchell & Plotkin (1988) were the first to connect the
of ML-style modules can be found in Chapter 2 of Russo’s thesis (1998; 2003).

The literature on ML module semantics is voluminous and varied. We will therefore
focus on the most closely related work. A more detailed history of various accounts
of ML-style modules can be found in Chapter 2 of Russo’s thesis (1998, 2003).

Related work and discussion

The literature on ML module semantics is voluminous and varied. We will therefore
focus on the most closely related work. A more detailed history of various accounts
of ML-style modules can be found in Chapter 2 of Russo’s thesis (1998, 2003).

Existential types for ADTs. Mitchell & Plotkin (1988) were the first to connect the
informal notion of “abstract type” to the existential types of System F. In F, values

Chapter 1 (The Design Space of ML Modules) of
Understanding and Evolving the ML Module System
(Dreyer, 2005)

Chapter 1

The Design Space of ML Modules

What is the ML module system? It is difficult to say. There are several dialects of the ML language,
and while the module systems of these dialects are certainly far more alike than not, there are
important and rather subtle differences among them, particularly with regard to the semantics of
data abstraction. The goal of Part I of this thesis is to offer a new way of understanding these
differences, and to derive from that understanding a unifying module system that harmonizes and
improves on the existing designs.

In this chapter, I will give an overview of the existing ML module system design space. I begin
in Section 1.1 by developing a simple example—a module implementing sets—that establishes some
basic terminology and illustrates some of the key features shared by all the modern variants of the
ML module system. Then, in Section 1.2, I describe several dialects that represent key points in
the design space, and discuss the major axes along which they differ.
A Type-Theoretic Approach to Higher-Order Modules with Sharing*

Robert Harper* Mark Lillibridge^ School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213-3881

Abstract

The design of a module system for constructing and maintaining large programs is a difficult task that raises a number of theoretical and practical issues. A fundamental issue is the management of the flow of information between program units at compile-time via the notion of an interface. Experience has shown that fully opaque interfaces are awkward to use in practice since too much information is hidden, and that fully transparent interfaces lead to excessive interdependence, creating problems for maintenance and separate compilation. The "sharing" specifications of Standard ML address this issue by allowing the programmer to specify equational relationships between types in separate modules, but are not expressive enough to allow the programmer complete control over the propagation of type information between modules.

These problems are addressed from a type-theoretic viewpoint by considering a calculus based on Girard's system F_
. The calculus differs from those considered in previous studies by relying exclusively on a new form of weak sum type to propagate information at compile-time, in contrast to approaches based on strong sums which rely on substitutions. The new form of sum type allows for the specification of equational relationships between types in separate modules and is sufficient to encode in a straightforward way most uses of type sharing specifications in Standard ML. Modules are treated as "first-class" citizens, and therefore the system supports higher-order modules and some object-oriented programming idioms; the language may be easily restricted to "second-class" modules found in ML-like languages.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title are given, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise or to republish, requires a fee and/or specific permission.

1 Introduction

Modularity is an essential technique for developing and maintaining large software systems [46, 24, 36]. Most modern programming languages provide some form of module system that supports the construction of large systems from a collection of separately-defined program units [7, 9, 26, 32]. A fundamental problem is the management of the tension between the need to treat the components of a large system in relative isolation (for both conceptual and pragmatic reasons) and the need to combine these components into a coherent whole.

Typical uses this problem is addressed by equipping each module with a well-defined interface that mediates all access to the module and requiring that interfaces be hidden at system link time.

The Standard ML (SML) module system [17, 32] is a particularly interesting design that has proved to be useful in the development of large software systems [2, 1, 3, 11, 13]. The main constituents of the SML module system are signatures, structures, and functors, with the latter two sometimes called modules.

A module is a program unit defining a collection of types, exceptions, values, and structures (known as substructures of the structure). A functor may be thought of as a "parameterized structure", a first-order function mapping structures to structures. A signature is an interface describing the constituents of a structure — the types, values, exceptions, and structures that it defines, along with their kinds, types, and interfaces. See Figure 1 for an illustrative example of the use of the SML module system: a number of sources are available for further examples and information [15, 36].

A crucial feature of the SML module system is the notion of equational sharing. This allows the programmer to specify that some constants should be shared across modules.

This work was sponsored by the Advanced Research Projects Agency, DARPA, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by SSD/AVS under Contract No. F19628-91-C-0168.

*Electronic mail address: mhd@cs.cmu.edu.

**bio: Robert Harper is an Associate Professor in the School of Computer Science at Carnegie Mellon University. His research interests include type systems for functional languages, parallel programming, and functional programming languages. He received his Ph.D. from the University of Edinburgh in 1984. He is a member of the ACM.

The language may be easily restricted to "second-class" modules found in ML-like languages.

123
Applicative functors and fully transparent higher-order modules

Xavier Leroy
INRIA
B.P. 105, Rouen, 76103 Le Chesnay, France.
Xavier.Leroy@inria.fr

Abstract

We present a variant of the Standard ML module system where parameterized abstract types (i.e., functors returning generative types) may provably equal arguments to compatible abstract types, instead of generating distinct types at each application as in Standard ML. This extension solves the full transparency problem (how to give syntactic signatures for higher-order functors that express exactly their propagation of type equations), and also provides better support for non-closed code fragments.

1 Introduction

Most modern programming languages provide support for type abstraction: the important programming technique where a named type is equipped with operations such as addition and multiplication. Type abstraction provides fundamental expression power, and justifies the recourse to complicated stamp-based representations of higher-order functors and separate compilation mechanisms. The work presented in this paper is an attempt to solve two of these problems (fully transparent higher-order functors and support for non-closed code fragments) in a syntactic framework derived from [10]. It relies on a modification of the behavior of functors (parameterized modules). In Standard ML and other models based on type generativity, a functor defines an abstract type returns a different type each time it is applied. We say that functors are parameterized if the functor is applied twice to provably equal arguments, the two abstract types returned remain compatible. Functors therefore may equally well express which allows equational reasoning on functor applications during type-checking. In turn, this allows more precise signatures for higher-order functors, thereby solving the full transparency problem.

Applicative functors are also interesting as an example of a module system that ensures type abstraction (the representation independence properties still hold) without requiring modulo type generativity (some applications of a given functor may return new types while others return compatible types). This is achieved through a careful choice of the representation independence properties (where are considered a semantic point of view (how are more programs of the same type (i.e., the semantics determined by the implementation?) rather than from an operational point of view (are two structurally identical types equal?)). This work illustrates the additional expressiveness and flexibility allowed by this shift of perspective.

“Reading 2: Applicative functors

We present a variant of the Standard ML module system where parameterized abstract types [...] map provably equal arguments to compatible abstract types, instead of generating distinct types at each application as in Standard ML.

“This extension solves the full transparency problem (how to give syntactic signatures for higher-order functors that express exactly their propagation of type equations)"
Abstract
ML modules are a powerful language mechanism for decomposing programs into reusable components. Unfortunately, they also have a reputation for being "complex" and requiring fancy type theory that is mostly opaque to non-experts. While this reputation is currently understandable, given the many non-standard methodologies that have been developed in the process of studying modules, we aim here to demonstrate that it is undeserved. To do so, we give a very simple elaboration semantics for a full-featured, higher-order ML-like module language. Our elaboration defines the meaning of module expressions by a straightforward, compositional translation into vanilla System $\mathsf{F}_\omega$, the higher-order polymorphic 3-calculus, under plain $\mathsf{F}_\omega$ typing environments. We thereby show that ML modules are merely a particular mode of use of System $\mathsf{F}_\omega$. Our module language supports the usual second-class constructs with Standard ML-style generative functions and module definitions. To demonstrate the viability of our approach, we further extend the language with the ability to package modules as first-class values—a very simple extension, as it turns out. Our approach also scales to handle OCaml-style applicative functor semantics, but the details are significantly more subtle, so we leave their presentation to a future, expanded version of this paper.

1. Introduction
Modularity is essential to the development and maintenance of large programs. Although most modern languages support modular programming and code reuse in one form or another, the languages in the ML family employ a particularly expressive style of module system. The key features shared by all the dialects of the ML module system are their support for functional (and imperative) sealing and implementor-side data abstraction (via translucent signatures). Although formally and conceptually elegant, their unifying account—which relies on singleton kinds, dependent types, and a type-theoretic type system—has, at least to our knowledge, never been fully explained, defining, studying, and evolving the ML module system is the primary source of the "complexity" complaint. Rather, we believe the problem is that the literature on the semantics of ML-style module systems is very dense and fragmented, to an outsider, it must surely be bewildering. Many non-standard type theoretic [18, 26, 25, 41, 9] (as well as several ad hoc, monstrosities [9, 31, 32]) methodologies have been developed for explaining, defining, and evolving the ML module systems, most with subtle semantic differences that are not spelled out clearly and are known only to experts. As a rich type theory has developed around a number of these methodologies—e.g., the beautiful metatheory of singleton kinds [45]—it is perfectly understandable for someone encountering a paper on module systems for the first time to feel intimidated by the apparent depth and breadth of knowledge required to understand module system semantics. Our elaboration defines the meaning of module expressions by a straightforward, compositional translation into vanilla System $\mathsf{F}_\omega$. We thereby show that ML modules are merely a particular mode of use of System $\mathsf{F}_\omega$. [O]ur approach is simpler and more accessible to someone who already understands $\mathsf{F}_\omega$ and does not want to learn a new dependent type system just in order to understand the semantics of ML modules.
Abstract types
How do approaches to abstract types differ between designs?

Separate compilation
How do ML-style modules systems support separate compilation?

Higher-order functors
Are higher-order functors practically important?

Importance of sharing
What is the role and significance of sharing specifications?

Dependent types vs polymorphism
Are modules better approached via dependent types or polymorphism?