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."

Modules for Standard ML (1984)
David MacQueen

Structures and signatures

a structure

```
module IntSet =
struct
  type elem = int
  type t = elem list
  let empty = []
  let rec mem x = function
   | [] → (* ... *)
end
```

a signature

```
module type SET  =
sig
  type elem
  type t
  val empty : t
  val mem : elem → t → bool
  (* ... *)
end
```

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).

a structure

```
module IntSet =
struct
  type elem = int
  type t = elem list
  let empty = []
  let rec mem x = function
   | [] → (* ... *)
end
```

a signature

```
module type SET  =
sig
  type elem
  type t
  val empty : t
  val mem : elem → t → bool
  (* ... *)
end
```

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).

Structures and signatures

a structure

```
module IntSet =
struct
  type elem = int
  type t = elem list
  let empty = []
  let rec mem x = function
   | [] → (* ... *)
end
```

a signature

```
module type SET  =
sig
  type elem
  type t
  val empty : t
  val mem : elem → t → bool
  (* ... *)
end
```

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).

Basics
History
Reading

a functor

```
module type ORDERED =
sig
  type t
  val compare : t → t → int
end

module MakeSet (Elem: ORDERED) =
struct
  type elem = Elem.t
  type t = elem list
  let mem = List.mem
  ...
end
```

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**.

Sharing as dependency

Module types involve various forms of **dependency**:

Dependency between **types** and **values**:

```
module type ORDERED =
sig
  type t
  val compare : t → t → int (* depends on t *)
end
```

Dependency between **arguments** and **results**:

```
module MakeSet :
  (Elem: ORDERED) →
    SET with type elem = Elem.t (* depends on Elem.t *)
```

Sharing as dependency

Module types involve various forms of **dependency**:

Dependency between **types** and **values**:

```
module type ORDERED =
sig
  type t
  val compare : t → t → int (* depends on t *)
end
```

Dependency between **arguments** and **results**:

```
module MakeSet :
  (Elem: ORDERED) →
    SET with type elem = Elem.t (* depends on Elem.t *)
```

Module types involve various forms of **dependency**:

Dependency between **types** and **values**:

```
module type ORDERED =
sig
  type t
  val compare : t → t → int (* depends on t *)
end
```

Dependency between **arguments** and **results**:

```
module MakeSet :
  (Elem : ORDERED) →
    SET with type elem = Elem.t (* depends on Elem.t *)
```

Higher-order modules

Using **higher-order modules** can lead to loss of type equalities:

higher-order functors

```
module Apply (MakeSet : (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

Modules and dependent types

1974 *Towards a theory of type structure* (Reynolds)

1985 *Abstract types have existential type* (Mitchell & Plotkin)

1986 *Using dependent types to express modular structure* (MacQueen)

1988 *The Calculus of Constructions* (Coquand & Huet)

1990 *Higher-order modules and the phase distinction* (Harper, Mitchell & Moggi)

1994 *A type-theoretic approach to higher-order modules with sharing* (Harper & Lillibridge)

dependent types

2010 *F-ing modules* (Rossberg, Russo & Dreyer)

Reading

Background reading (optional)

§11 (*Related work and discussion*) of
F-ing modules, extended version
(Rossberg, Russo, Dreyer, 2015)

11 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)

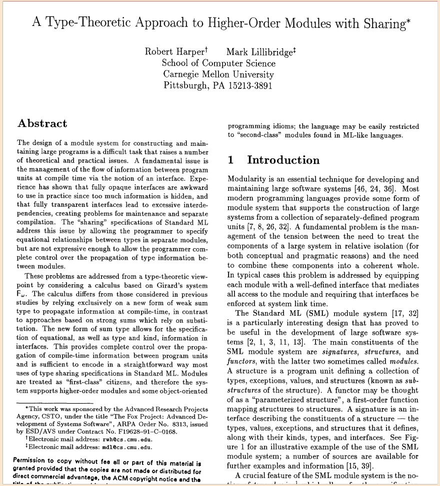
Chapter 1

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.

Paper 1: Translucent sums



A Type-Theoretic Approach to Higher-Order Modules with Sharing*

Robert Harper† Mark Lillibridge‡
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

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 interdependencies, 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_ω. 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 substitution. The new form of sum type allows for the specification of equational, as well as type and kind, information in interfaces. This provides complete control over the propagation of compile-time information between program units 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.

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, 8, 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. In typical cases 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 enforced 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 structure 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, 39].

A crucial feature of the SML module system is the no-

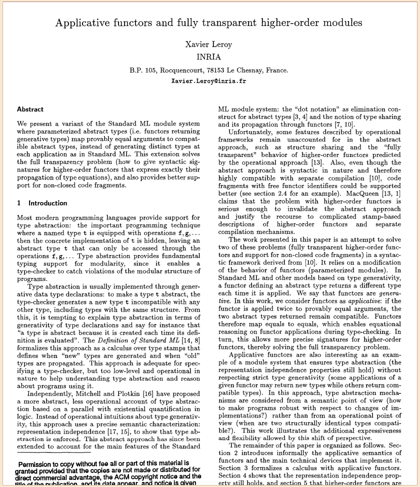
*This work was sponsored by the Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628–91–C–0168.

†Electronic mail address: rwh@cs.cmu.edu.
‡Electronic mail address: mdl@cs.cmu.edu.

"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 substitution [...]

"Modules are treated as "first-class" citizens, and therefore the system supports higher-order modules and some object-oriented programming idioms"

Paper 2: Applicative functors



Applicative functors and fully transparent higher-order modules

Xavier Leroy
INRIA
B.P. 105, Rocquencourt, 78153 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) 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), 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 `t` is equipped with operations `f, g, ...` then the concrete implementation of `t` is hidden, leaving an abstract type `t` that can only be accessed through the operations `f, g, ...` Type abstraction provides fundamental typing support for modularity, since it enables a type-checker to catch violations of the modular structure of programs.

Type abstraction is usually implemented through generative data type declarations: to make a type `t` abstract, the type-checker generates a new type `t` incompatible with any other type, including types with the same structure. From this, it is tempting to explain type abstraction in terms of generativity of type declarations and say for instance that "a type is abstract because it is created each time its definition is evaluated". The *Definition of Standard ML* [14, 8] formalizes this approach as a calculus over type stamps that defines when "new" types are generated and when "old" types are propagated. This approach is adequate for specifying a type-checker, but too low-level and operational in nature to help understanding type abstraction and reason about programs using it.

Independently, Mitchell and Plotkin [16] have proposed a more abstract, less operational account of type abstraction based on a parallel with existential quantification in logic. Instead of operational intuitions about type generativity, this approach uses a precise semantic characterization: representation independence [17, 15], to show that type abstraction is enforced. This abstract approach has since been extended to account for the main features of the Standard ML module system: the "dot notation" as elimination construct for abstract types [3, 4] and the notion of type sharing and its propagation through functors [7, 10].

Unfortunately, some features described by operational frameworks remain unaccounted for in the abstract approach, such as structure sharing and the "fully transparent" behavior of higher-order functors predicted by the operational approach [13]. Also, even though the abstract approach is syntactic in nature and therefore highly compatible with separate compilation [10], code fragments with free functor identifiers could be supported better (see section 2.4 for an example). MacQueen [13, 1] claims that the problem with higher-order functors is serious enough to invalidate the abstract approach and justify the recourse to complicated stamp-based descriptions 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 defining an abstract type returns a different type each time it is applied. We say that functors are *generative*. In this work, we consider functors as *applicative*: if the functor is applied twice to provably equal arguments, the two abstract types returned remain compatible. Functors therefore map equals to equals, which enables 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 respecting strict type generativity (some applications of a given functor may return new types while others return compatible types). In this approach, type abstraction mechanisms are considered from a semantic point of view (how to make programs robust with respect to changes of implementations?) rather than from an operational point of view (when are two structurally identical types compatible?). This work illustrates the additional expressiveness and flexibility allowed by this shift of perspective.

The remainder of this paper is organized as follows. Section 2 introduces informally the applicative semantics of functors and the main technical devices that implement it. Section 3 formalizes a calculus with applicative functors. Section 4 shows that the representation independence property still holds, and section 5 that higher-order functors are

"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)"

Paper 3: F-ing modules

F-ing Modules

Andreas Rossberg
MPI-SWS
rossberg@mpi-sws.org

Claudio V. Russo
Microsoft Research
crusso@microsoft.com

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

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 certainly 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 F_ω (the higher-order polymorphic λ-calculus), under plain F_ω typing environments. We thereby show that ML modules are merely a particular mode of use of System F_ω.

Our module language supports the usual second-class modules with Standard ML-style generative functors and local module definitions. To demonstrate the versatility 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.

Lastly, we report on our experience using the "locally nameless" approach in order to mechanize the soundness of our elaboration semantics in Coq.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, Abstract data types; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Design, Theory

Keywords Type systems, ML modules, abstract data types, existential types, System F, elaboration, first-class modules

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 hierarchical namespace management (via *structures*), a fine-grained variety of interfaces (via *translucent signatures*), client-side data abstraction (via *functors*), and implementor-side data abstraction (via *sealing*).

Unfortunately, while the utility of ML modules is not in dispute, they have nonetheless acquired a reputation for being "complex". Simon Peyton Jones, in an oft-cited POPL 2003 keynote address [35], likened ML modules to a Porsche, due to their "high power, but poor power/cost ratio". (In contrast, he likened Haskell—extended with various "sexy" type system extensions—to a Ford Cortina with alloy wheels.) Although we disagree with Peyton Jones' amusing analogy, it seems, based on conversations with many others in the field, that the view that ML modules are too complex for mere mortals to understand is sadly predominant.

Why is this so? Are ML modules really more difficult to program/implement/understand than other ambitious modularity mechanisms, such as GHC's *type classes* with type equality coercions [44] or Java's *classes* with generics and wildcards [45]? We think not (although this is obviously a fundamentally subjective question). One can certainly engage in a constructive debate about whether the mechanisms that comprise the ML module system are put together in the ideal way, and in fact the first and third authors have recently done precisely that [11]. But we do not believe that the *design* of 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 so vast and fragmented that, to an outsider, it must surely be bewildering. Many non-standard type-theoretic [18, 16, 26, 25, 41, 9] (as well as several *ad hoc*, non-type-theoretic [30, 31, 3]) methodologies have been developed for explaining, defining, studying, 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 [43]—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 typechecking, let alone module compilation.

In response to this problem, Dreyer, Crary and Harper [9] developed a unifying type theory, in which previous systems can be understood as sublanguages that selectively include different features. Although formally and conceptually elegant, their unifying

"Our elaboration defines the meaning of module expressions by a straightforward, compositional translation into vanilla System F_ω [...] We thereby show that ML modules are merely a particular mode of use of System F_ω. [...]

"[T]he previous [translations] all start from a pre-existing dependently-typed module language and show how to compile it down to F_ω [...] [O]ur approach is simpler and more accessible to someone who already understands F_ω and does not want to learn a new dependent type system just in order to understand the semantics of ML modules."

Writing suggestions

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?