

A module system independent of base languages

Hyeonseung Im^{1,2} and Sungwoo Park^{1,3}

*Department of Computer Science and Engineering
Pohang University of Science and Technology
Republic of Korea*

Abstract

The ML module system facilitates modular programming and data abstraction using nested modules, higher-order functors, and abstract types. However, it is difficult to adapt to various base languages due to specific features required for supporting abstract types across module boundaries and the assumption that the base language consists of terms and types. This paper proposes a module system that is highly independent of the base language while providing nested modules and higher-order functors. In order to maximize the independence between the module system and the base language, we assume that the base language consists of abstract declarations and specifications rather than terms and types. Furthermore, we allow references to components of modules only via module paths. This paper discusses the current design of our module system building on these assumptions and outlines future work.

Keywords: Module System, Path Resolution Problem, Path Conversion, Defunctorization

1 Introduction

The ML module system [11,6,8,9,16,1] facilitates modular programming and data abstraction using *nested modules*, *higher-order functors*, and *abstract types*. *Structures*, or *modules*, are collections of related *declarations* such as definitions of datatypes and associated operations. *Functors*, parameterized modules, are functions from structures to structures. *Signatures* and *functor signatures* are called *module types* and specify interfaces to structures and functors. Nested modules allow modules as components, higher-order functors take functors as arguments, and abstract types hide the implementation details of types. Combinations of these constructs provide better support for flexible program construction, information hiding, and code reuse.

¹ The authors are grateful to anonymous reviewers for their helpful comments. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF), grant number R11-2008-007-03001-0.

² Email: genilhs@postech.ac.kr

³ Email: gla@postech.ac.kr

The ML module system consists of two linguistic levels: a *base* (or *core*) *language* and a *module language*. The base language defines terms and types to be used inside modules and module types, while the module language provides modular programming constructs such as nested modules and higher-order functors.

Although it provides powerful support for modular programming, the ML module system is difficult to adapt to various base languages for two reasons. First, it mainly focuses on supporting abstract types across module boundaries [6,8,9,16,1] rather than on ensuring that the module system is independent of the base language. Abstract types are by-products of the interaction between the module and base languages, and thus the ML module system requires specific features for supporting abstract types. For example, the Definition of Standard ML [14] represents the identities of abstract types using stamps, i.e. unique names, at the base language level. The type-theoretic interpretation of Harper and Stone [7], which translates Standard ML into an internal language, shows that stamps are not necessary. However, redesigning their interpretation independently of the base language is unclear, although they conjecture that such languages as Caml, Haskell, and Scheme could be translated into a similar internal language. As another example, the Objective Caml module system [15] requires the base language to provide a mapping from *access-paths*, i.e. references to components of modules, to types and a type matching relation between access-paths. Second, most previous work on the ML module system assumes that the base language consists of terms and types extended with access-paths. Therefore, if the base language includes additional constructs such as processes and data-flow graphs as in [13] or needs other forms of specifications such as logical properties as in [4], the module system should be extended or redesigned accordingly.

In this paper, we propose a module system that is highly independent of the base language while providing nested modules and higher-order functors. In order to maximize the independence between the module system and the base language, we use the following assumptions:

- From the perspective of the module language, the base language consists of abstract declarations and specifications rather than terms and types. Thus, we may incorporate additional constructs into the base language without extending or redefining the module system. The structure of base language environments is also abstract at the module level, whereas in most previous work it is defined at the module level. Hence, we can define the static semantics for the base language almost independently of the module system.
- From the perspective of the base language, *paths*, sequences of module names, are the only visible module-level constructs. Hence the interaction between the module and base languages (e.g. references to components of modules) is allowed only via paths.
- We assume that the entire source code is available so that we can apply *defunctorization* and *demodularization* [5]. Defunctorization reduces a functor application into a module derived from the functor body. Demodularization transforms nested modules into flat modules and remove (higher-order) functors. By defunctorization and demodularization, we transform a source program into a base language

<pre> 1 module T = struct 2 module X = struct 3 type s = int 4 module G (A: sig type t end) = 5 struct 6 type u = s 7 module B = struct 8 type v = A.t * u 9 end 10 end 11 end 12 module C = struct type t = X.s end 13 module R = X.G(C) 14 end </pre>		<pre> 1 module T : sig 2 module X : sig 3 type s = int 4 module G : functor (A: sig type t end) -> 5 sig 6 type u = s 7 module B : sig 8 type v = A.t * u 9 end 10 end 11 end 12 module C : sig type t = X.s end 13 module R : sig ... end 14 end </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 1. Example module and its signature

program, which implies that we can continue to use the dynamic semantics for the base language. As for typechecking, we may separately typecheck each module before applying defunctorization and demodularization.

The main focus here is on supporting as many base languages as possible while ensuring that extending or modifying the base language does not affect the module system. Ideally, our module system aims to accommodate any base language. The downside of our module system is that we can support only a restricted form of abstract types (if the base language includes abstract type specifications such as `type t` in Objective Caml).

The rest of the paper is organized as follows. Section 2 presents a technical challenge due to nested modules along with our approach to the challenge. Section 3 presents assumptions on the base language. Section 4 presents the current design of our module system. Section 5 discusses related work and outlines future work.

2 Technical Challenge

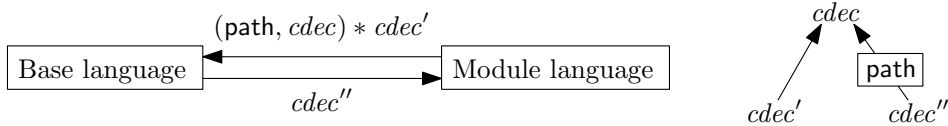
In the presence of nested modules, scoping rules are as follows. First, the scope of each component of a module extends from its declaration point to the end of the module. Second, components from previously defined modules are referred to via paths of the modules relative to the current location. Consider the (rather contrived) example written in Objective Caml in Figure 1. The left side defines a module named `T` while the right side specifies its signature. According to the scoping rules, type `s` declared in line 3 is referred to by using access-path `s` in line 6, while it is referred to by using access-path `X.s` in line 12.

The scoping rules cause some difficulty, which we call a *path resolution problem*, when we define the static semantics for the module language and defunctorization. The path resolution problem states that access-paths may be valid or invalid depending on their locations. For example, in order to typecheck functor application `X.G(C)` in line 13, we first look up the signature of functor `X.G` (lines 4–10). Note that specification `type u = s` (line 6) in the signature refers to type `s` declared earlier in module `X` by using access-path `s`. However, after line 11, type `s` must be referred to by using access-path `X.s`. Hence, if we directly use the signature of

$X.G$, the signature of $X.G(C)$ will contain access-path s which is invalid at line 13. Likewise, the same problem occurs when eliminating functor application $X.G(C)$ by substituting actual argument C for formal argument A in the functor body.

An explicit substitution-based approach [8,9,16] solves the path resolution problem. As an example, consider eliminating functor application $X.G(C)$ in line 13. Since type s declared in module X is referred to by using access-path s in the definition of functor $X.G$ (lines 4–10), we apply substitution $\{s \mapsto X.s\}$ to the functor definition, thus making every access-path in the functor definition valid at line 13. Then, substituting actual argument C for formal argument A in the functor body gives the result of eliminating $X.G(C)$.

However, our module system cannot directly adopt the explicit substitution-based approach because base language declarations, specifications, and environments are all abstract at the module level. Instead, the module language delegates to the base language the task of generating and applying substitutions. More precisely, we require the base language to provide an implicit substitution operation that rewrites access-paths so that base language components are referred to only via a module path specified by the module language. For example, in the following figure, base language declaration $cdec'$ (such as in line 6) refers to another declaration $cdec$ (such as in line 3) as if they were in the same module. When the module language provides a path $path$ associated with $cdec$, the base language returns $cdec''$ equivalent to $cdec'$ except that $cdec''$ refers to $cdec$ via the given path $path$.



To solve the path resolution problem using the implicit substitution operation, we are led to define *path conversion* that converts *relative paths* to *full paths*. Full paths are paths that start from the top-level module name and thus valid everywhere, while relative paths are paths that obey the two scoping rules mentioned earlier. After path conversion, components of modules are referred to only via full paths. For example, if module T is the top-level module, then path conversion converts type $u = s$ (line 6) into type $u = T.X.s$. Path conversion, however, cannot convert access-path u in line 8 because the full path for the functor body is unknown. Hence path conversion rewrites every functor so that it takes an additional argument for full paths for the results of its applications. If variable Z ranges over full paths, then path conversion now converts functor G as follows:

```

module G (Z1 : sig type t end, Z2) = struct
  type u = T.X.s
  module B = struct type v = Z1.t * Z2.u end
end

```

Here, path variable Z_2 ranges over full paths for the results of applications of functor G . Note that we also convert argument A into path variable Z_1 .

3 Base Language *BL*

Figure 2 shows our assumptions on the base language *BL*. Throughout the paper, we write in *italic* for the definition of the base language and in sans serif for the

definition of the module system.

		environment	CE
module path	\mathbf{q}	environment union	$CE \uplus CE'$
declaration	$cdec$	well-formed declaration	$CE \vdash cdec \Rightarrow CE'$
specification	$cspec$	well-formed specification	$CE \vdash cspec \Rightarrow CE'$
		environment subsumption	$CE \vdash CE_1 \leq CE_2$
(1) Abstract syntax		(2) Semantic objects and judgments	

Fig. 2. Assumptions on base language BL

At the syntactic level, we assume that BL consists of abstract declaration $cdec$ and specification $cspec$. A base language program is a single $cdec$ while an interface is a single $cspec$. ($cdec$ may include a list of declarations such as a series of type and function definitions. Similarly for $cspec$.) With the introduction of the module system, a $cdec$ (or $cspec$) may refer to $cdec$'s (or $cspec$'s) defined in other modules. References to those $cdec$'s defined in other modules are allowed only via module paths \mathbf{q} which are abstract at the base language level. Module paths are the only module-level constructs visible to BL .

At the semantic level, we assume that the static semantics for BL interprets a $cdec$ (or $cspec$) as an environment CE whose definition is left unspecified. \uplus is an associative union operator for CE 's. We read judgments of the form $CE \vdash cdec \Rightarrow CE'$ “ $cdec$ elaborates to CE' under CE ” (similarly for $cspec$). We also assume that the static semantics provides a subsumption relation $CE \vdash CE_1 \leq CE_2$ which reads “ CE_1 is subsumed by CE_2 under CE ”. We say that a $cdec$ conforms to, or implements, a $cspec$ if the following three conditions hold: 1) $CE \vdash cdec \Rightarrow CE_1$, 2) $CE \vdash cspec \Rightarrow CE_2$, and 3) $CE \vdash CE_1 \leq CE_2$.

We have identified three operations that the base language should provide:

- $[\] :: \mathbf{q} \rightarrow CE \rightarrow CE'$

Instead of using module environments at the base language level, we require an operation $[\]$ that combines a module path \mathbf{q} and a base language environment CE to produce an extended environment CE' that associates \mathbf{q} with each component in CE . Specifically, for any base language construct t , the specification of t from CE is the same as the specification of $\mathbf{q}.t$ from CE' (if the base language uses the standard dot notation for module component access). We write $[\mathbf{q}]$ for the application of $[\]$ to \mathbf{q} .

- $\zeta :: (\mathbf{q}_1, CE_1), \dots, (\mathbf{q}_n, CE_n) \rightarrow CE \rightarrow cdec \rightarrow cdec'$

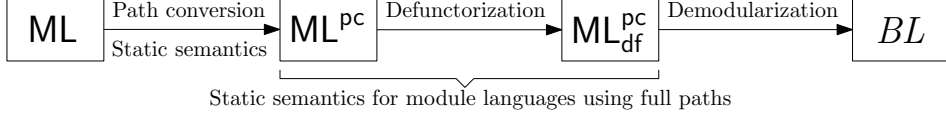
ζ is a generalization of the implicit substitution operation introduced in Section 2. Declarations in submodules refer to declarations defined earlier in surrounding modules as if they were defined in the same module (e.g. access-path \mathbf{s} in `type u = s` in Figure 1). ζ changes those references so that they are referred to via full paths of surrounding modules. \mathbf{q}_1 is the full path for the outermost surrounding module while \mathbf{q}_n is for the innermost surrounding module. CE is the environment for declarations defined earlier in the same module where $cdec$ is defined. ζ

similarly applies to a *cspec*.

- Given that a path substitution $\psi = \{q_i \mapsto q'_i \mid i = 1, \dots, n\}$ is provided by the module system, the base language should provide an operation that applies ψ to *cdec*, *cspec*, or *CE*. We write $\psi(\text{cdec})$ for the declaration obtained by applying the substitution ψ to every path that appears in *cdec* (similarly for *cspec* and *CE*).

4 Module System

This section discusses the current design of our module system.



The above figure shows the overall structure of our module system. We define the module system as three translations from module language *ML* to base language *BL*. Source language *ML* provides nested modules and higher-order functors. Intermediate languages ML^{pc} and ML^{pc}_{df} are the target languages for path conversion and defunctorization, respectively. Base language *BL* is the target language for demodularization. We define both the static semantics for *ML* and path conversion from *ML* to ML^{pc} in a single translation. Therefore, the static semantics accepts module expressions as well-typed modules if and only if their path conversion succeeds. Defunctorization translates ML^{pc} into ML^{pc}_{df} which is a subset of ML^{pc} . To be specific, defunctorization eliminates each functor application by substituting actual arguments for formal arguments in the functor body. ML^{pc} and ML^{pc}_{df} use a static semantics different from that for *ML* because they use full paths instead of relative paths. Demodularization translates ML^{pc}_{df} into *BL* by eliminating all other remaining module-level constructs.

Figure 3 shows the abstract syntax of *ML* and ML^{pc} . We use \circ to denote constructs of ML^{pc} . In *ML*, path *p* denotes relative paths while variable *x* ranges over module names. In ML^{pc} , constant *t* denotes the top-level module name while variable *z* ranges over full paths. In both *ML* and ML^{pc} , functor applications use only paths instead of general module expressions. Abstract declarations and specifications are the only base language constructs visible to *ML* and ML^{pc} . In ML^{pc} , a functor `functor` ($z_1 : M^\circ, z_2$) $\rightarrow m^\circ$ uses variable z_2 for full paths for the results of its applications. Consequently, a functor application $p^\circ_1(p^\circ_2, p^\circ_3)$ specifies that the resultant module is located at full path p°_3 .

We define the static semantics as an elaboration process from syntactic objects into semantic objects as in [14]. Figure 4 (1) shows semantic objects for *ML*. A base typing *CT* carries a full path p° and the base language environment *CE* for p° . *L* denotes an ordered list CT_1, \dots, CT_n where CT_1 is for the outermost surrounding module (top-level module) of the module being elaborated while CT_{n-1} is for the innermost surrounding module. CT_n is for the module being elaborated. Functor types *F* are dependent types augmented with full paths of functors. For simplicity, we assume that functors take and return only structures as arguments and results. To use higher-order functors, functors should be wrapped inside structures.

path	$p ::= x \mid p.x$	full path	$p^\circ ::= t \mid z \mid t.p \mid z.p$
module expression	$m ::= p \mid \text{struct } d_1, \dots, d_n \text{ end}$ $\mid \text{functor } (x : M) \rightarrow m$ $\mid p_1(p_2) \mid (m : M)$	converted module expression	$m^\circ ::= p^\circ \mid \text{struct } d_1^\circ, \dots, d_n^\circ \text{ end}$ $\mid \text{functor } (z_1 : M^\circ, z_2) \rightarrow m^\circ$ $\mid p_1^\circ(p_2^\circ, p_3^\circ) \mid (m^\circ : M^\circ)$
module type	$M ::= \text{sig } D_1, \dots, D_n \text{ end}$ $\mid \text{functor } (x : M_1) \rightarrow M_2$	converted module type	$M^\circ ::= \text{sig } D_1^\circ, \dots, D_n^\circ \text{ end}$ $\mid \text{functor } (z_1 : M_1^\circ, z_2) \rightarrow M_2^\circ$
declaration	$d ::= \text{cdec} \mid \text{module } x = m$	converted declaration	$d^\circ ::= \text{cdec}^\circ \mid \text{module } x = m^\circ$
specification	$D ::= \text{cspec} \mid \text{module } x : M$	converted specification	$D^\circ ::= \text{cspec}^\circ \mid \text{module } x : M^\circ$
	(1) Syntax of ML		(2) Syntax of ML ^{PC}

 Fig. 3. Syntax of module languages ML and ML^{PC}

set of z's	$V = \{z_1, \dots, z_n\}$	m and M conversion	$E \vdash m : T \Rightarrow m^\circ \quad E \vdash M : T \Rightarrow M^\circ$
base typing	$CT = (p^\circ, CE)$	d and D conversion	$E \vdash d : S \Rightarrow d^\circ \quad E \vdash D : S \Rightarrow D^\circ$
list of CT's	$L = CT_1, \dots, CT_n$	subsumption relation	$E \vdash T_1 \leq T_2$
module typing	$MT = \{x \mapsto T\}$	typing conversion	$L \rightsquigarrow CE \quad MT \rightsquigarrow \psi, CE'$
structure type	$S = CT; MT$		
functor type	$F = (p^\circ, \Pi z_1 : S_1. S_2)$		
module type	$T = S \mid F$		
module env.	$E = V; L; MT$		
	(1) Semantic objects for ML		(2) Judgments for ML

Fig. 4. Semantic objects and judgments for module language ML

Figure 4 (2) shows a subset of judgments that we use for the static semantics for ML and path conversion from ML to ML^{PC}. The module conversion judgment $E \vdash m : T \Rightarrow m^\circ$ means that m is of module type T and converted into m° . (Similarly for M , d and D .) The subsumption relation judgment $E \vdash T_1 \leq T_2$ means that T_1 is a subtype of T_2 . Rules for $E \vdash T_1 \leq T_2$ use the base language judgment $CE \vdash CE_1 \leq CE_2$. The typing conversion judgments $L \rightsquigarrow CE$ and $MT \rightsquigarrow \psi, CE'$ mean that L and MT convert into CE and CE' , respectively. Conversion uses the

top-level module

$$\frac{\emptyset; (t, \emptyset); \{\} \vdash \text{struct } d_1, \dots, d_n \text{ end} : S \Rightarrow \text{struct } d^{\circ}_1, \dots, d^{\circ}_n \text{ end}}{E \vdash \text{module } t = \text{struct } d_1, \dots, d_n \text{ end} : \{t \mapsto S\} \Rightarrow \text{module } t = \text{struct } d^{\circ}_1, \dots, d^{\circ}_n \text{ end}} \quad (1)$$

module expressions ($E \vdash m : T \Rightarrow m^{\circ}$)

$$\frac{(\text{MT of } E)(x) = T \quad \text{full_path}(T) = p^{\circ}}{E \vdash x : T \Rightarrow p^{\circ}} \quad (2) \quad \frac{E \vdash p : S \Rightarrow p^{\circ} \quad (\text{MT of } S)(x) = T \quad \text{full_path}(T) = p^{\circ'}}{E \vdash p.x : T \Rightarrow p^{\circ'}} \quad (3)$$

$$\frac{E \vdash S_1 + \dots + S_{i-1} \vdash d_i : S_i \Rightarrow d^{\circ}_i \quad (1 \leq i \leq n)}{E \vdash \text{struct } d_1, \dots, d_n \text{ end} : S_1 + \dots + S_n \Rightarrow \text{struct } d^{\circ}_1, \dots, d^{\circ}_n \text{ end}} \quad (4)$$

$$\frac{E \oplus (\{z_1\}; (z_1, \emptyset); \{\}) \vdash M : S_1 \Rightarrow M^{\circ} \quad E \oplus (\{z_1, z_2\}; (z_2, \emptyset); \{x \mapsto S_1\}) \vdash m : S_2 \Rightarrow m^{\circ}}{E \vdash \text{functor } (x : M) \rightarrow m : (p^{\circ}, \Pi z_1 : S_1 . S_2) \Rightarrow \text{functor } (z_1 : M^{\circ}, z_2) \rightarrow m^{\circ}} \quad (5)$$

$$\frac{E \vdash p_1 : (p^{\circ}_1, \Pi z_1 : S_1 . S_2) \Rightarrow p^{\circ}_1 \quad E \vdash p_2 : S'_1 \Rightarrow p^{\circ}_2 \quad E \vdash S'_1 \leq S_1}{E \vdash p_1(p_2) : \{z_1 \mapsto p^{\circ}_2\} S_2 \Rightarrow p^{\circ}_1(p^{\circ}_2, \text{cur_path}(E))} \quad (6)$$

$$\frac{E \vdash m : T_1 \Rightarrow m^{\circ} \quad E \vdash M : T_2 \Rightarrow M^{\circ} \quad E \vdash T_1 \leq T_2}{E \vdash (m : M) : T_2 \Rightarrow (m^{\circ} : M^{\circ})} \quad (7)$$

- $\text{full_path}((p^{\circ}, CE); \text{MT}) = p^{\circ}$, $\text{full_path}(p^{\circ}, \Pi z_1 : S_1 . S_2) = p^{\circ}$, $\text{cur_path}(V; L, (p^{\circ}, CE); \text{MT}) = p^{\circ}$
- $V; L, (p^{\circ}, CE); \text{MT} + (p^{\circ}, CE'); \text{MT}' = V; L, (p^{\circ}, CE \uplus CE'); \text{MT} + \text{MT}'$
where $\text{MT} + \text{MT}'$ is a usual composition of mappings.
- $V; L; \text{MT} \oplus V'; (p^{\circ'}, CE'); \text{MT}' = V \cup V'; L, (p^{\circ'}, CE'); \text{MT} + \text{MT}'$

declarations ($E \vdash d : S \Rightarrow d^{\circ}$)

$$\frac{E \oplus (\emptyset; (p^{\circ}.x, \emptyset); \{\}) \vdash m : T \Rightarrow m^{\circ} \quad p^{\circ} = \text{cur_path}(E) \quad T' = \{\text{full_path}(T) \mapsto p^{\circ}.x\}T}{E \vdash \text{module } x = m : (p^{\circ}, \emptyset); \{x \mapsto T'\} \Rightarrow \text{module } x = m^{\circ}} \quad (8)$$

$$\frac{L \rightsquigarrow CE \quad \text{MT} \rightsquigarrow \psi, CE' \quad CE_n \uplus [p^{\circ}_n] CE_n \uplus CE \uplus CE' \vdash \zeta(L, CE_n, \psi(cdec)) \Rightarrow CE''}{V; L, (p^{\circ}_n, CE_n); \text{MT} \vdash cdec : (p^{\circ}_n, CE''); \{\} \Rightarrow \zeta(L, CE_n, \psi(cdec))} \quad (9)$$

Fig. 5. Inference rules for the static semantics for ML and path conversion from ML to ML^{PC}

\square operation provided by the base language. $\text{MT} \rightsquigarrow \psi, CE'$ additionally generates path substitution ψ which replaces relative paths with their full paths.

Figure 5 shows selected inference rules for the static semantics for ML and path conversion from ML to ML^{PC}. Inference rules for signatures, functor signatures, and specifications are similar to those for structures, functors, and declarations, respectively. In rules (5) and (8), $\{p^{\circ} \mapsto p^{\circ'}\}T$ substitutes $p^{\circ'}$ for every occurrence of p° in T . In rule (9), we elaborate base language declarations after converting all relative paths to corresponding full paths. That is, we elaborate $\zeta(L, CE_n, \psi(cdec))$ instead of $cdec$ where the ζ operation is provided by the base language.

Except for using full paths, the static semantics for ML^{PC} and ML^{PC}_{df} is similar to that for ML. Thanks to path conversion, defunctorization is easily defined: a functor application $p^{\circ}_1(p^{\circ}_2, p^{\circ}_3)$ is reduced to the body of functor p°_1 where formal arguments are replaced with actual arguments p°_2 and p°_3 .

5 Related Work and Future Work

The ML module system, originally proposed by MacQueen [11], has been extensively studied due to its powerful support for modular programming [6,8,9,16,1]. Among

others, Leroy [10] proposes an ML-style module system that makes few assumptions about the base language in order to accommodate a variety of base languages. Including his module system, however, most previous work mainly focuses on supporting abstract types, thus requiring specific features in the base language. In contrast, we trade abstract types for the independence between the module system and the base language so that the module system can accommodate as many base languages as possible. Still, we support a restricted form of abstract types, which we believe are useful enough in practice.

Our module system is similar to RTG type system of Dreyer for recursive modules [2,3] in that a functor takes an additional argument as input. In RTG, the additional argument represents the names of abstract types to be defined inside the functor body, whereas in our module system, the additional argument represents full paths for the results of functor applications. In the module system of Nakata and Garrigue for recursive modules [12], each structure has an additional variable, called *self variable*, through which components of the structure refer to each other recursively. Therefore, every functor body also has its own self variable. In their module system, all applications of a functor to the same argument share a common self variable because functor applications are all applicative as in [9]. In contrast, our module system instantiates the additional argument to a functor with a different full path at each functor application.

Defunctorization and demodularization in our work were inspired by static interpretation of modules proposed by Elsmann [5] which also eliminates all module-level constructs at compile time. His interpretation, however, builds on an ML-style base language while we take an approach independent of the base language.

We have outlined a module system independent of the base language, which provides nested modules and higher-order functors. We impose only a few assumptions on the base language. So far, we have designed module languages ML , ML^{PC} , and $\text{ML}_{\text{df}}^{\text{PC}}$ along with path conversion, defunctorization, and two static semantics (one for ML , the other for ML^{PC} and $\text{ML}_{\text{df}}^{\text{PC}}$). Currently, we are proving the soundness of path conversion and defunctorization, and designing demodularization. After finishing the design of our module system, we can translate any program with nested modules and higher-order functors into a base language program without module-level constructs.

References

- [1] Dreyer, D., K. Crary, and R. Harper, *A type system for higher-order modules*, In *POPL '03*, 236–249.
- [2] Dreyer, D., *Recursive Type Generativity*, In *ICFP '05*, 41–53.
- [3] Dreyer, D., *A Type System for Recursive Modules*, In *ICFP '07*, 289–302.
- [4] Eastlund, C., and M. Felleisen, *Toward a Practical Module System for ACL2*, In *PADL '09*, 46–60.
- [5] Elsmann, M., *Static interpretation of modules*, In *ICFP '99*, 208–219.
- [6] Harper, R., and M. Lillibridge, *A type-theoretic approach to higher-order modules with sharing*, In *POPL '94*, 123–137.
- [7] Harper, R., and C. Stone, *A type-theoretic interpretation of Standard ML*, In “Proof, Language, and Interaction: Essays in Honour of Robin Milner,” MIT Press, 2000.

- [8] Leroy, X., *Manifest types, modules, and separate compilation*, In *POPL '94*, 109–122.
- [9] Leroy, X., *Applicative functors and fully transparent higher-order modules*, In *POPL '95*, 142–153.
- [10] Leroy, X., *A modular module system*, *Journal of Functional Programming* **10(3)** (2000), 269–303.
- [11] MacQueen, D., *Modules for Standard ML*, In *LFP '84*, 198–207.
- [12] Nakata, K., and J. Garrigue, *Recursive Modules for Programming*, In *ICFP '06*, 74–86.
- [13] Nowak, D., J. Talpin, T. Gautier, and P. L. Guernic, *An ML-like Module System for the Synchronous Language SIGNAL*, In *Euro-Par '97*, 1244–1253.
- [14] Milner, R., M. Tofte, R. Harper, and D. MacQueen, “The Definition of Standard ML (Revised),” MIT Press, 1997.
- [15] Objective Caml, <http://caml.inria.fr/>.
- [16] Shao, Z., *Transparent modules with fully syntactic signatures*, In *ICFP '99*, 220–232.