Typed Cross-Module Compilation*

Zhong Shao
Dept. of Computer Science
Yale University
New Haven, CT 06520
shao-zhong@cs.yale.edu

Abstract

Higher-order modules are very effective in structuring large programs and defining generic, reusable software components. Unfortunately, many compilation techniques for the core languages do not work across the module boundaries. As a result, few optimizing compilers support these module facilities well.

This paper exploits the semantic property of ML-style modules to support efficient cross-module compilation. More specifically, we present a type-directed translation of the MacQueen-Tofte higher-order modules into a predicative variant of the polymorphic λ-calculus F_ω. Because modules can be compiled in the same way as ordinary polymorphic functions, standard type-based optimizations such as representation analysis immediately carry over to the module languages.

We further show that the full-transparency property of the MacQueen-Tofte system yields a near optimal cross-module compilation framework. By propagating various static information through the module boundaries, many static program analyses for the core languages can be extended to work across higher-order modules.

1 Introduction

Modular programming has proven to be extremely valuable in the development and maintenance of large software systems [3, 31, 9]. Many modern programming languages such as Modula-3 [31] and Standard ML [26, 27] provide support for both core-level and module-level programming. The core language, in general, deals with the detailed implementation of algorithms in terms of data structures and control constructs. The module language, on the other hand, provides glue to organize large programs and to build generic and reusable components. A *mature* and *scalable* compiler must support both styles of programming well, generating decent code even for heavily modularized programs.

ML-style higher-order modules [25, 11, 20] are widely recognized as one of the most powerful module constructs in existence today. Recent work on the type-theoretic foundations of ML modules [11, 19, 27] has cleaned up many rough spots in the original design [26]. Still, the semantics for higher-order modules involves the use of dependent types [24, 12] or translucent signatures [11, 19]. MacQueen and Tofte [25] have shown that even a small restriction on signature matching [19, 11] can significantly compromise the overall expressiveness (i.e., full transparency) of the underlying module language. It is fair to say that the type systems for higher-order modules is much more elaborate than (or at least very different from) those for the core-ML-like languages [8].

This semantic difference between the core and module languages poses great challenges to compiler writers. Although the module code itself seldom needs to be compiled efficiently, optimizations used for the core language must be compatible with the module constructs in order to have a coherent compiler. Unfortunately, many compilation techniques do not work on programs that use higher-order modules. In fact, most recent work on compiling functional languages have ignored issues on the necessary module support. Take the area of *type-directed compilation* as an example: recent work includes representation analysis [18, 33], type specialization [6], intensional type analysis [14], typed closure conversion [28], tagless garbage collection [40], to name just a few. All of these are performed on the variants of core ML [8] or the polymorphic λ-calculus F_ω [10, 32]. While they have all demonstrated that types can be used to make programs run faster and consume less space, it is not obvious how any of these would work in the presence of higher-order modules (which make use of dependent types).

Consider the module code in Figure 1, written in the SML syntax. Here, structure `SA`, `SB`, and `SC` are simple modules; signature `ASIG` and `BSIG` express the interface of structures; functor `F` (inside `SB`) and `APP`, also known as parameterized modules, are just functions from structures to structures. Functor specifications such as `F` inside `BSIG` use functor signatures to express the interface of functors.

*This research was sponsored in part by the Defense Advanced Research Projects Agency ITO under the title "Software Evolution using HOT Language Technology," DARPA Order No. D888, issued under Contract No. F30602-96-2-0232, and in part by an NSF CAREER Award CCR-9501624, and NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

To appear in ICFP'98: Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming, September 1998, Baltimore, Maryland.

```
signature ASIG =
  sig type s
      val f : s
  end

signature BSIG =
  sig functor F(X : ASIG) : ASIG
  end

structure SA =
  struct type s = int
         val f = 3
  end
```

```
structure SB =
  struct
    functor F(X : ASIG) =
      struct
        type s = X.s -> X.s
        fun f (x : X.s) = X.f
      end
  end

functor APP (B : BSIG) = B.F(SA)

structure SC = APP(SB)
```

Figure 1: An example of ML-style higher-order modules

Structure `SB` and functor `APP` are examples of the so-called "higher-order" modules: structures (`SB`) can contain other functors as their components, and functors (`APP`) can take arbitrary structures as their arguments. Functors can only be applied to structures with compatible signatures, as in `APP(SB)`; the result structure has an interface like that of the original functor body, but with proper instantiations.

It is easy to see why higher-order modules would break the type-based optimizations mentioned above. In the F_ω calculus, polymorphic functions such as the identity function $\Lambda t::\Omega.\lambda x:t.x$ always cleanly separate the type abstraction (Λ) from the value abstraction (λ). Therefore, polymorphic functions in F_ω or core ML can be *specialized* to particular type arguments at compile time. Furthermore, both representation analysis [18, 33] and intensional type analysis [14] can be performed, inserting coercions or runtime type parameters at every type-application site. Higher-order modules, on the other hand, express both the type and value abstractions through a single construct (i.e., functor). A functor such as `APP` and `SB.F` takes mixed sets of types and values as its argument, and return another such set as its result. Functor applications such as `APP(SB)` cannot be type-specialized, because we do not know how to identify the exact type parameters in functors such as `APP`. Representation analysis and intensional type analysis are also hard to perform because of the pervasive use of dependent types.

Higher-order modules also make it very difficult to carry out static program analysis across the module boundaries. Because the module signature does not propagate any static information other than types, many existing techniques, such as constant propagation, function inlining [2], partial evaluation [17], and constraint-based analysis [1], lose all their information at the functor-application boundaries. In the previous example, if we textually inline all functor applications in the source, we can deduce that the `f` component in structure `SC` is equivalent to the following:

```
fun f (x : int) = 3
```

However, because inlining large functors can lead to code explosion, and moreover, modules often must be compiled separately, it is impractical to eliminate all functors by inlining. The challenge then is to deduce these properties statically while still supporting separate compilation.

This paper exploits the semantic property [13] of ML-style modules to support efficient cross-module compilation. More specifically, we present a type-directed translation of the MacQueen-Tofte higher-order modules [25] into a predicative variant of the F_ω calculus. Because modules can be compiled in the same way as ordinary polymorphic functions, all the type-based optimizations mentioned above immediately carry over to the module languages. The basic idea of our algorithm is similar to phase-splitting [13]: we notice that every ML module can be split into a *type* part and a *value* part; the type (value) part of a structure includes all of its type (value) components plus the type (value) parts of its structure and functor components; the type part of a functor is a higher-order type function from the type part of its arguments to that of its result; the value part of a functor can be viewed as a polymorphic function quantified over the type part of its arguments; functor applications can thus be expressed as a combination of type application and value application as in the F_ω calculus.

We further show that the *full-transparency* property of the MacQueen-Tofte system yields a near optimal cross-module compilation framework. Here, by full transparency, we mean that type information is always propagated optimally through all the module boundaries, so structures such as `SC` get exactly the same typing whether functor `APP` is textually inlined or separately compiled. By propagating other static information in the same way, we can extend most static program analyses for core languages to work across higher-order modules.

The main contributions of this paper are:

- As far as we know, our work is the first comprehensive and formal study on how to apply type-based compilation techniques [18, 40, 14, 28, 33] to programs using ML-style modules. Our main result that ML-style modules can be compiled into an F_ω-like calculus is new and significant because immediately all type-based techniques for F_ω become applicable to the module languages as well.
- Our translation of the MacQueen-Tofte system into the F_ω calculus is the first such algorithm that deals with the essential features in the ML-like module languages. Several recent papers [13, 4, 20] have attacked similar problems but with completely different motivations; they also impose severe restrictions to their module languages (e.g., no type abbreviation or sharing inside signatures [13], no parameterized types [4],

$$
\begin{array}{llll}
\textit{kind} & \kappa_t & ::= & \Omega \mid \kappa_t \to \kappa'_t \mid \{l :: \kappa_t, ..., l' :: \kappa'_t\} \\
\textit{tycon} & \mu_t & ::= & \alpha \mid \texttt{Int} \mid \mu_t \to \mu'_t \mid \lambda\alpha :: \kappa_t.\mu_t \mid \mu_t[\mu'_t] \mid \{l = \mu_t, ..., l' = \mu'_t\} \mid \mu_t.l \\
\textit{type} & \sigma_t & ::= & T(\mu_t) \mid \sigma_t \to \sigma'_t \mid \{l : \sigma_t, ..., l' : \sigma'_t\} \mid \forall\alpha :: \kappa_t.\sigma_t \\
\textit{term} & e_t & ::= & x \mid i \mid \lambda x : \sigma_t.e_t \mid @e_t e'_t \mid \Lambda\alpha :: \kappa_t.e_t \mid e_t[\mu_t] \mid \{l = e_t, ..., l' = e'_t\} \mid e_t.l \mid \texttt{let}\ d_t\ \texttt{in}\ e_t \\
\textit{decl} & d_t & ::= & \varepsilon \mid (x = e_t); d_t
\end{array}
$$

Figure 2: Syntax of the F_ω-based target calculus TGC

$$
\frac{\Delta \triangleright \mu_t :: \kappa_t \quad ... \quad \Delta \triangleright \mu'_t :: \kappa'_t}{\Delta \triangleright \{l = \mu_t, ..., l' = \mu'_t\} :: \{l :: \kappa_t, ..., l' :: \kappa'_t\}}\ (cprod) \qquad \frac{\Delta \triangleright \mu_t :: \{..., l :: \kappa_t, ...\}}{\Delta \triangleright \mu_t.l :: \kappa_t}\ (cselect)
$$

$$
\frac{\Delta \triangleright \{..., l = \mu_t, ...\} :: \{..., l :: \kappa_t, ...\}}{\Delta \triangleright \{..., l = \mu_t, ...\}.l \equiv_t \mu_t :: \kappa_t}\ (cp1{-}eqv) \qquad \frac{\Delta \triangleright \mu_t :: \kappa''_t \quad \kappa''_t = \{l :: \kappa_t, ..., l' :: \kappa'_t\}}{\Delta \triangleright \{l = \mu_t.l, ..., l' = \mu_t.l'\} \equiv_t \mu_t :: \kappa''_t}\ (cp2{-}eqv)
$$

$$
\frac{\Delta; \Gamma \vdash\!\!\!^{\mu} \ e_t : \sigma_t \quad ... \quad \Delta; \Gamma \vdash\!\!\!^{\mu} \ e'_t : \sigma'_t}{\Delta; \Gamma \vdash\!\!\!^{\mu} \ \{l = e_t, ..., l' = e'_t\} : \{l : \sigma_t, ..., l' : \sigma'_t\}}\ (prod) \qquad \frac{\Delta; \Gamma \vdash\!\!\!^{\mu} \ e_t : \{..., l : \sigma_t, ...\}}{\Delta; \Gamma \vdash\!\!\!^{\mu} \ e_t.l : \sigma_t}\ (select)
$$

Figure 3: Selected typing rules for TGC

and limited forms of functor arguments [20]).

- Our compilation algorithm can handle the entire SML'97 language [27] including both *transparent* and *opaque* signature matching. In fact, the algorithm has been implemented and released with the SML/NJ compiler since version 109.24 (January 9, 1997). As a result, all type-based optimizations in the compiler work across the higher-order module boundaries.
- We also describe a new algorithm that does both cross-module inlining and type specialization, even for functions with free value and type variables. Our algorithm supports fully transparent propagation of binding information, even across heavily functorized code. Other kinds of program analysis can be extended to work across higher-order modules in the same way.
- To facilitate our presentation, we give a new and more complete formal definition for the MacQueen-Tofte higher-order modules. MacQueen and Tofte's original semantics [25] does not address many important features such as type specifications, type declarations, and hidden module components. Our new semantics covers a much richer language and solves the remaining technical problems.

The rest of this paper is organized as follows: we first define an F_ω-based target calculus (TGC) and an ML-style higher-order module calculus (NRC). The NRC calculus contains all the essential features in an ML-style module system. We give the static semantics for NRC and then present a type-directed translation from NRC to TGC. We show how to exploit the full-transparency property to support cross-module program analysis. Finally, we discuss implementation details, related work, and then conclude.

2 An F_ω-based target calculus

Our target calculus TGC is a predicative variant [13] of the polymorphic λ-calculus F_ω. The syntax of TGC is given in Figure 2. Here, kinds classify type constructors (*tycon*); types classify terms. Declarations (*dec*) and the term $\texttt{let}\ d_t\ \texttt{in}\ e_t$ are syntactic sugar introduced to simplify the presentation of our translation algorithm. Constructors of kind Ω name monotypes. The monotypes are generated from variables, `Int`, and through the arrow constructor ($\to$). The application and abstraction constructors correspond to the function kind $\kappa_t \to \kappa'_t$. The product and selection constructors correspond to the product kind $\{l :: \kappa_t, ..., l' :: \kappa'_t\}$. Types in TGC include the monotypes, and are closed under products, function spaces, and polymorphic quantification. Following Harper and Morrisett [14], we use $T(\mu)$ to denote the corresponding monotype of the constructor μ. The terms are an explicitly typed λ-calculus with explicit constructor abstraction (Λ) and application forms. TGC terms also include the labeled product and selection terms; product terms such as $\{l = e_t, ..., l' = e'_t\}$ correspond to product types of $\{l : \sigma_t, ..., l' : \sigma'_t\}$.

The static semantics for TGC consists of the following set of typing rules:

constructor formation	$\Delta \triangleright \mu_t :: \kappa_t$
constructor equivalence	$\Delta \triangleright \mu_t \equiv_t \mu'_t :: \kappa_t$
type formation	$\Delta \triangleright \sigma_t$
type equivalence	$\Delta \triangleright \sigma_t \equiv_t \sigma'_t$
term formation	$\Delta; \Gamma \vdash\!\!\!^{\mu} \ e_t : \sigma_t$
declaration formation	$\Delta; \Gamma \vdash\!\!\!^{\mu} \ d_t : \Gamma'$

Here, Δ is a kind environment that maps from type variables to kinds, and Γ is a type environment that maps from program variables to types. The constructor formation rule is of form $\Delta \triangleright \mu_t :: \kappa_t$, meaning that constructor μ_t has kind κ_t under kind environment Δ. Two constructors μ_t and μ'_t of kind κ_t are considered equivalent if $\Delta \triangleright \mu_t \equiv_t \mu'_t :: \kappa_t$. The term formation rule is of form $\Delta; \Gamma \vdash\!\!\!^{\mu} \ e_t : \sigma_t$, meaning that term e_t is assigned type σ_t under kind environment Δ and type environment Γ. Figure 3 gives the typing rules for the product-related constructors and terms. The rest of the rules are all standard, as shown in the companion technical report [35]. It is well known that type-checking for the TGC-like calculus is decidable, and furthermore, its typing rules are consistent with properly defined operational semantics [29, 14].

Access paths:

strpath	p_s	::=	$s_i \mid p_s.s_i$
fctpath	p_f	::=	$f_i \mid p_s.f_i$
typpath	p_t	::=	$t_i \mid p_s.t_i$

Signatures and specifications:

spec	D	::=	$\varepsilon \mid DD'$
		$\mid$	`type` $t_i :: \kappa_c$
		$\mid$	`type` $t_i :: \kappa_c = \mu_c$
		$\mid$	`structure` $s_i : M_s$
		$\mid$	`functor` $f_i : M_f$
sig	M_s	::=	`sig` D `end`
fsig	M_f	::=	`fsig`$(s_i : M_s) M'_s$

Kinds and types:

kind	κ_c	::=	$\Omega \mid \Omega \rightarrow \kappa_c$
tycon	μ_c	::=	$p_t \mid int \mid \mu_c \rightarrow \mu'_c$
		$\mid$	$\lambda t_i :: \Omega.\mu_c \mid \mu_c[\mu'_c]$

Modules and declarations:

decl	d	::=	$\varepsilon \mid dd' \mid$ `local` d `in` d' `end`
		$\mid$	`type` $t_i :: \kappa_c = \mu_c$
		$\mid$	`structure` $s_i = m_s$
		$\mid$	`functor` $f_i = m_f$
str	m_s	::=	$p_s \mid f_i(s_i) \mid (s_i : M_s) \mid m_b$
	m_b	::=	`struct` d `end`
fct	m_f	::=	$p_f \mid$ `funct` $(s_i : M_s) m_b$

Figure 4: Syntax of the normalized module calculus NRC

3 Module semantics and translation

To show how we translate the ML-style modules into the TGC calculus, we choose a surface language (SFC) that is similar to the SML'97 module language[27] extended with the MacQueen-Tofte higher-order modules [25]. We divide the translation from SFC to TGC into two steps:

- First, we perform a series of syntactic transformations and normalizations, and then translate the SFC program into a normalized module calculus (NRC);
- Second, we translate the NRC program into the TGC calculus.

To make the paper more focused, we will only briefly explain the first step, and then concentrate on describing the NRC calculus and how we translate NRC into the TGC calculus. The companion TR [35] gives more details about the SFC calculus and the SFC-to-NRC translation.

3.1 Normalization and the NRC calculus

The syntax of our normalized module calculus NRC is defined in Figure 4. During the SFC-to-NRC translation, each identifier in the surface language is alpha-converted and assigned a unique internal name, so an identifier (e.g., x_i) in NRC always consists of two parts: an external name (x) and an internal stamp (i). We use t, s, and f to denote type, structure, and functor identifiers, and p_t, p_s, and p_f for the access paths.

The NRC module language includes standard constructs such as signatures (M_s), functor signatures (M_f), structure expression (m_s), and functor expressions (m_f). A signature contains a list of type, structure, and functor specifications (D). A type specification can either be a *flexible* one—specifying only the arity of the underlying type constructor; or it can be a type abbreviation. Type-sharing specifications in SFC are converted into type abbreviations [22]. A structure expression can be a structure path (p_s), a functor application ($f_i(s_i)$), a structure matched by a signature ($s_i : M_s$), or a definitional structure (m_b). Functors in NRC are higher-order because they can be passed as arguments or returned as results by other functors.

To simplify the presentation, we did not include value specification and value declaration in the current NRC calculus. We also used a rather simple type language where μ_c denotes type constructor and κ_c denotes its kind (i.e, the arity). These simplifications do not affect the generality of our algorithm since the main idea of our translation can be illustrated independent of these core-language features. Section 5 gives more details on how to handle value components and other advanced core-language features.

The translation from SFC to NRC does a series of normalizations that turn complex SFC constructs into simpler and more primitive NRC constructs. The most important transformation is to make explicit all *enrichment* coercions occurring at every signature matching (see the companion TR [35] for details). After this transformation, a structure s_i in NRC will only match a signature M_s if the following are all satisfied: first, s_i must contain the same number of components as those in M_s; second, each pair of corresponding components in s_i and M_s must have the same external and internal names and follow the same order; finally, each functor component in s_i must have a signature syntactically equivalent to its counterpart in M_s. These invariants are reflected and enforced in NRC's static semantics given in Section 3.2.

Another unusual feature of NRC is that functor body must be of form `struct...end`, i.e., a definitional structure expression marked syntactically as m_b. This is exploited by the static semantics to ensure a one-to-one mapping between type stamps and their definitional type paths. To convert an arbitrary functor expression (say `funct` $(s_i : M_s) m_s$) into this form, the SFC-to-NRC translation wraps an extra layer around the functor body m_s; the resulting functor body becomes:

`funct` $(s_i : M_s)$ `struct structure` R `=` m_s `end.`

Functor bodies inside functor signatures are transformed in the same way. Meanwhile, functor application such as $F(S)$ is translated into:

`let structure` $T = F(S)$ `in` $T.R$ `end.`

Finally, all module declarations inside the `let` expressions in SFC are pushed upwards and then turned into the NRC `local` declarations (see the companion TR [35] for details).

Stamp	n	$\equiv$	same as i	*Path*	p_x	$::=$	$p_s \mid \varepsilon$
Tycon	μ_m	$::=$	$n(\kappa_c, \mu_t) \mid \alpha \mid int \mid \mu_m \rightarrow \mu'_m$	*StampEnv*	N	$\equiv$	$Stamp \xrightarrow{fin} TypPath$
		$\mid$	$\lambda\alpha :: \Omega.\mu_m \mid \mu_m[\mu'_m]$	*SpecEnv*	D	$\equiv$	$Id \xrightarrow{fin} Spec$
TypRlzn	r_t	$::=$	μ_m	*RlznEnv*	R	$\equiv$	$Id \xrightarrow{fin} Rlzn$
StrRlzn	r_s	$::=$	R	*Basis*	B	$\equiv$	(Δ, N, D, R)
FctRlzn	r_f	$::=$	$(m_b, B, A_t) \mid (A_t)$	*AuxInfo*	A_t	$::=$	(μ_t, σ_t)
Rlzn	r	$::=$	$r_t \mid r_s \mid r_f$				

Figure 5: Semantic objects for NRC

signature subsumption	$B \vdash^s M_s \leq M'_s$	Figure 7
module declaration	$B \vdash^d d : N ; D ; R \Longrightarrow d_t$	Figure 7
module expression	$B \vdash^m m\_ : N ; M\_ ; r\_ \Longrightarrow e_t$	Figure 7
signature instantiation	$\mu_t ; p_x ; B \vdash^I M_s : N ; r_s \Longrightarrow \sigma_t$	Figure 8
signature kind translation	$\vdash^\kappa M\_ \Longrightarrow \kappa_t$	Figure 9
module type translation	$\vdash^\kappa (M\_, r\_) \Longrightarrow \mu_t ; \sigma_t$	Figure 9

Figure 6: NRC semantics and translation: a summary

3.2 Static semantics for NRC

Before presenting the translation from NRC to TGC, we first give a new and more complete formal semantics for the MacQueen-Tofte higher-order modules [25] in the context of NRC. Under our stamp-based approach, the "type" of a module—also called the *modtype* in this paper—is expressed as a pair of a signature and a *realization*. The signature captures the module skeleton such as names of its components and the kind and sharing information for type specifications. The realization describes the actual type definition for all the type paths (p_t) in an NRC construct. The job of the static semantics is to infer and validate the semantic types for all NRC type paths, and then propagate this information to different modules.

Figure 5 defines the semantic objects used by our new semantics. Figure 6 gives a summary of all the semantic rules; we use a single set of deduction rules to describe both the static semantics and the translation algorithm. A deduction such as $W \vdash X : Y \Longrightarrow Z$ has the following meaning: under the environment W, the NRC construct X is elaborated into the semantic object Y and translated into the TGC construct Z. In the rest of this paper, we use $\emptyset_N$, $\emptyset_R$, etc. to denote the empty environments; $\uplus$ to denote the environment overlay; $\bullet$ to denote things that are irrelevant under the current rule; and $s_i.p_x$ to denote an access path that starts with an structure s_i and ends with a tail path p_x. We also restrict our semantics to elaborate *normalized* NRC programs only; an NRC program is normalized if there are no duplicate bindings within each scope, and all of its functor definitions have distinctly named formal parameters; these conditions are enforced by the SRC-to-NRC translation.

Returning to Figure 5, here, μ_m is an internal "semantic" type constructor designed for type-checking. The only difference between μ_m and the NRC type constructor μ_c is that μ_c might be a type path p_t but μ_m cannot be. During the elaboration, formal type constructors in NRC are translated into type stamps of the form $n(\kappa_c, \mu_t)$ where n is a stamp, κ_c is its kind, and μ_t is a TGC type constructor—auxiliary information used solely for the NRC-to-TGC translation. The type equivalence relation $\equiv_m$ on μ_m is the standard structural equivalence except that two stamped types $n(\kappa_c, \mu_t)$ and $n'(\kappa'_c, \mu'_t)$ are equivalent if and only if n and n' are equal.

A realization can be a type realization (r_t), a structure realization (r_s), or a functor realization (r_f). A type realization captures the actual definition of a type component; it is represented simply as the internal type constructor μ_m. A structure realization captures the detailed definitions of all the components in a structure; it is defined as a *realization environment* (R) which maps from (type, structure, and functor) identifiers to realizations. A functor realization captures the typing relationship between the argument and the result of a functor; it is defined either as a *realization closure* (m_b, B, A_t) or as a *formal template* (A_t). In both cases, A_t contains auxiliary TGC type information maintained solely for the NRC-to-TGC translation.

The realization for a fully defined functor, e.g., `funct` $(s_i : M_s)m_b$, is a realization closure (m'_b, B, A_t). The code part of the closure, m'_b, is simply the actual functor body m_b. The environment part of the closure is the current basis[1] B (defined below). The realization for a formal functor parameter, e.g., functor `B.F` inside `APP` in Figure 1, is defined as a formal template, marked as (A_t); all we know about such functor is its signature.

The basis environment B is a tuple (Δ, N, D, R) where Δ is an auxiliary TGC kind environment, N is a stamp environment, D is a specification environment (represented as NRC specifications), and R is a realization environment. The kind environment Δ does not play any role for the static semantics; it is purely maintained for the NRC-to-TGC translation (mainly to simplify the technical proof). The stamp environment N records all the type stamps defined so far and maps each of them to its definitional type path. The specification environment D and the realization environment R form the actual modtype environment. Given a module access path $p\_$, we can retrieve its modtype by looking it up in the corresponding environments; the result is abbreviated as $B(p\_) = (M\_, r\_)$ where wild card "_" implies either structure entity (s) or functor entity (f).

At any time during the elaboration, an NRC type μ_c can be mapped into its actual semantic type μ_m, and vice versa.

[1] Neither the kind environment nor the stamp environment is required here, but we include it anyway to simplify the notation.

Signature subsumption: $\boxed{B \vdash^{s} M_s \leq M_s'}$

$$\frac{B \vdash^{s} D \leq D'}{B \vdash^{s} \texttt{sig}\ D\ \texttt{end} \leq \texttt{sig}\ D'\ \texttt{end}}\ (1) \qquad \frac{B \vdash^{s} D \leq D'' \quad B \vdash^{s} D' \leq D'''}{B \vdash^{s} DD' \leq D''D'''}\ (2)$$

$$B \vdash^{s} \varepsilon \leq \varepsilon\ (3) \qquad B \vdash^{s} \texttt{functor}\ f_i : M_f \leq \texttt{functor}\ f_i : M_f\ (4)$$

$$\frac{B \vdash^{s} M_s \leq M_s'}{B \vdash^{s} \texttt{structure}\ s_i : M_s \leq \texttt{structure}\ s_i : M_s'}\ (5) \qquad \frac{\mathbf{c2m}(B, \mu_c) \equiv_m \mathbf{c2m}(B, \mu_c')}{B \vdash^{s} \texttt{type}\ t_i :: \kappa_c = \mu_c \leq \texttt{type}\ t_i :: \kappa_c = \mu_c'}\ (6)$$

$$B \vdash^{s} \texttt{type}\ t_i :: \kappa_c = \mu_c \leq \texttt{type}\ t_i :: \kappa_c\ (7) \qquad B \vdash^{s} \texttt{type}\ t_i :: \kappa_c \leq \texttt{type}\ t_i :: \kappa_c\ (8)$$

Typing and translation of declaration(s): $\boxed{B \vdash^{d} d : N ; D ; R \Longrightarrow d_t}$

$$\frac{\mu_m = \mathbf{c2m}(B, \mu_c) \qquad \mu_m \text{ has kind } \kappa_c}{B \vdash^{d} \texttt{type}\ t_i :: \kappa_c = \mu_c : \emptyset_{\mathrm{N}} ; \{\texttt{type}\ t_i :: \kappa_c = \mu_c\} ; \{t_i \mapsto \mu_m\} \Longrightarrow \varepsilon}\ (9)$$

$$\frac{B \vdash^{m} m_s : N ; M_s ; r_s \Longrightarrow e_t \qquad N' = \{n \mapsto s_i.p_t \mid p_t = N(n), n \in \mathrm{Dom}(N)\}}{B \vdash^{d} \texttt{structure}\ s_i = m_s : N' ; \{\texttt{structure}\ s_i : M_s\} ; \{s_i \mapsto r_s\} \Longrightarrow (s_i = e_t)}\ (10)$$

$$\frac{B \vdash^{m} m_f : N' ; M_f ; r_f \Longrightarrow e_t \qquad N' \equiv \emptyset_{\mathrm{N}}}{B \vdash^{d} \texttt{functor}\ f_i = m_f : \emptyset_{\mathrm{N}} ; \{\texttt{functor}\ f_i : M_f\} ; \{f_i \mapsto r_f\} \Longrightarrow (f_i = e_t)}\ (11)$$

$$B \vdash^{d} \varepsilon : \emptyset_{\mathrm{N}} ; \emptyset_{\mathrm{D}} ; \emptyset_{\mathrm{R}} \Longrightarrow \varepsilon\ (12)$$

$$\frac{B \vdash^{d} d' : N' ; D' ; R' \Longrightarrow d_t' \qquad B \uplus (\emptyset_{\Delta}, N', D', R') \vdash^{d} d'' : N'' ; D'' ; R'' \Longrightarrow d_t''}{B \vdash^{d} d'd'' : N' \uplus N'' ; D'D'' ; R' \uplus R'' \Longrightarrow d_t'd_t''}\ (13)$$

$$\frac{B \vdash^{d} d' : N' ; D' ; R' \Longrightarrow d_t' \qquad B \uplus (\emptyset_{\Delta}, N', D', R') \vdash^{d} d'' : N'' ; D'' ; R'' \Longrightarrow d_t''}{B \vdash^{d} \texttt{local}\ d'\ \texttt{in}\ d''\ \texttt{end} : N' \uplus N'' ; D'' ; R' \uplus R'' \Longrightarrow d_t'd_t''}\ (14)$$

Typing and translation of module expression: $\boxed{B \vdash^{m} m\_ : N ; M\_ ; r\_ \Longrightarrow e_t}$

$$\frac{B(p\_) = (M\_, r\_)}{B \vdash^{m} p\_ : \emptyset_{\mathrm{N}} ; M\_ ; r\_ \Longrightarrow p\_}\ (15) \qquad \frac{B(s_i) = (M_s', r_s) \qquad B \vdash^{s} M_s' \leq M_s}{B \vdash^{m} (s_i : M_s) : \emptyset_{\mathrm{N}} ; M_s ; r_s \Longrightarrow s_i}\ (16)$$

$$\frac{B \vdash^{d} d : N' ; D' ; R' \Longrightarrow d_t \qquad e_t = \{x_i = x_i, \ldots\} \text{ for all } x_i \in \mathrm{Dom}(D'),\ x_i \text{ not a type}}{B \vdash^{m} \texttt{struct}\ d\ \texttt{end} : N' ; \texttt{sig}\ D'\ \texttt{end} ; R\mathrm{of}B \uplus R' \Longrightarrow \texttt{let}\ d_t\ \texttt{in}\ e_t}\ (17)$$

$$\frac{\begin{array}{c}\vdash^{x} M_s \Longrightarrow \kappa_t \qquad \Delta = \{s_i :: \kappa_t\} \qquad s_i ; s_i ; B \uplus (\Delta, \emptyset_{\mathrm{N}}, \emptyset_{\mathrm{D}}, \emptyset_{\mathrm{R}}) \vdash^{I} M_s : N ; r_s \Longrightarrow \sigma_t \\ B \uplus (\Delta, N, \{\texttt{structure}\ s_i : M_s\}, \{s_i \mapsto r_s\}) \vdash^{m} m_b : \bullet ; M_s' ; r_s' \Longrightarrow e_t \\ \vdash^{x} (M_s', r_s') \Longrightarrow \mu_t' ; \sigma_t' \qquad A_t = (\lambda s_i :: \kappa_t.\mu_t', \quad \forall s_i :: \kappa_t.\sigma_t \to \sigma_t') \qquad r_f = (m_b, B, A_t)\end{array}}{B \vdash^{m} \texttt{funct}\ (s_i : M_s) m_b : \emptyset_{\mathrm{N}} ; \texttt{fsig}(s_i : M_s) M_s' ; r_f \Longrightarrow \Lambda s_i :: \kappa_t.\lambda s_i : \sigma_t.e_t}\ (18)$$

$$\frac{\begin{array}{c}B(f_i) = (M_f, r_f) \qquad B(s_i) = (M_s, r_s) \qquad M_f = \texttt{fsig}(s_i' : M_s) M_s' \\ D' = \{\texttt{structure}\ s_i' : M_s\} \qquad R' = \{s_i' \mapsto r_s\} \qquad \vdash^{x} (M_s, r_s) \Longrightarrow \mu_t ; \bullet \\ \text{if } r_f = (A_t) \text{ and } A_t = (\mu_t', \bullet) \text{ then } \mu_t'[\mu_t] ; \varepsilon ; B \uplus (\emptyset_{\Delta}, \emptyset_{\mathrm{N}}, D', R') \vdash^{I} M_s' : N' ; r_s' \Longrightarrow \bullet \\ \text{if } r_f = (m_b', B', \bullet) \text{ then } (\Delta\mathrm{of}B, N\mathrm{of}B, D\mathrm{of}B' \uplus D', R\mathrm{of}B' \uplus R') \vdash^{m} m_b' : N' ; \bullet ; r_s' \Longrightarrow \bullet\end{array}}{B \vdash^{m} f_i(s_i) : N' ; M_s' ; r_s' \Longrightarrow @(f_i[\mu_t])s_i}\ (19)$$

Figure 7: Module semantics and its translation into TGC

Signature instantiation and translation: $\boxed{\mu_t\,;p_x\,;B \vdash M_s\ :\ N\,;\,r_s \Longrightarrow \sigma_t}$

$$\frac{\mu_t\,;p_x\,;B \vdash D\ :\ N'\,;\,R' \Longrightarrow lts \qquad R'' = R\mathrm{of}B \uplus R'}{\mu_t\,;p_x\,;B \vdash \texttt{sig}\ D\ \texttt{end}\ :\ N'\,;\,R'' \Longrightarrow \{lts\}} \quad (20)$$

$$\frac{\mu_t\,;p_x\,;B \vdash D\ :\ N\,;\,R \Longrightarrow lt \qquad \mu_t\,;p_x\,;B \uplus (\emptyset_\Delta, N, D, R) \vdash D'\ :\ N'\,;\,R' \Longrightarrow lts}{\mu_t\,;p_x\,;B \vdash DD'\ :\ N \uplus N'\,;\,R \uplus R' \Longrightarrow lt, lts} \quad (21)$$

$$\mu_t\,;p_x\,;B \vdash \varepsilon\ :\ \emptyset_{\mathrm{N}}\,;\,\emptyset_{\mathrm{R}} \Longrightarrow \varepsilon \quad (22) \qquad \frac{\mu_m = \mathbf{c2m}(B, \mu_c) \quad \mu_m \text{ has kind } \kappa_c \quad R = \{t_i \mapsto \mu_m\}}{\mu_t\,;p_x\,;B \vdash (\texttt{type}\ t_i :: \kappa_c = \mu_c)\ :\ \emptyset_{\mathrm{N}}\,;\,R \Longrightarrow \varepsilon} \quad (23)$$

$$\frac{n \notin \mathrm{Dom}(N\mathrm{of}B) \qquad \kappa_t = \kappa_c \qquad \mu_m = n(\kappa_c, \mu_t.t_i)}{\mu_t\,;p_x\,;B \vdash (\texttt{type}\ t_i :: \kappa_c)\ :\ \{n \mapsto p_x.t_i\}\,;\,\{t_i \mapsto \mu_m\} \Longrightarrow \varepsilon} \quad (24)$$

$$\frac{\mu_t.s_i\,;p_x.s_i\,;B \vdash M_s\ :\ N\,;\,r_s \Longrightarrow \sigma_t}{\mu_t\,;p_x\,;B \vdash (\texttt{structure}\ s_i : M_s)\ :\ N\,;\,\{s_i \mapsto r_s\} \Longrightarrow s_i : \sigma_t} \quad (25)$$

$$\frac{\begin{array}{lll} \vdash^{\omega} M_s \Longrightarrow \kappa_t & \Delta = \{s_i :: \kappa_t\} & s_i\,;s_i\,;B \uplus (\Delta, \emptyset_{\mathrm{N}}, \emptyset_{\mathrm{D}}, \emptyset_{\mathrm{R}}) \vdash M_s\ :\ N\,;\,r_s \Longrightarrow \sigma_t \\ \mu_t'' = \mu_t.f_i & \mu_t''[s_i]\,;\varepsilon\,;B \uplus (\Delta, N, \{\texttt{structure}\ s_i : M_s\}, \{s_i \mapsto r_s\}) \vdash M_s'\ :\ \bullet\ ;\bullet \Longrightarrow \sigma_t' & \\ \sigma_t'' = \forall s_i :: \kappa_t.\sigma_t \to \sigma_t' & A_t = (\mu_t'', \sigma_t'') & r_f = (A_t) \end{array}}{\mu_t\,;p_x\,;B \vdash (\texttt{functor}\ f_i : \texttt{fsig}(s_i : M_s) M_s')\ :\ \emptyset_{\mathrm{N}}\,;\,\{f_i \mapsto r_f\} \Longrightarrow f_i : \sigma_t''} \quad (26)$$

Figure 8: Signature instantiation and its translation into TGC

Given a basis environment $B = (N, D, R)$, the **c2m** operator converts μ_c into a semantic type by replacing each type path p_t in μ_c with its actual definition $R(p_t)$; the result is denoted as $\mathbf{c2m}(B, \mu_c)$. Similarly, the **m2c** operator can convert a semantic type μ_m back to the external format by replacing each stamped type $n(\kappa_c, \mu_t)$ with its definitional type path $N(n)$. The **m2c** operator is not used in the current semantics, but it is useful for inferring the signature of modules with value (i.e., `val`) components.

The most unusual aspect of our semantics is the rules for signature subsumption (see Rules 1–8 in Figure 7): they are much more restrictive than those used by MacQueen and Tofte [25]. To have one signature subsume another, both must contain the same number of components, following the same order (Rules 2 and 3); furthermore, the respective functor components must have syntactically equivalent signature (Rule 4). This restriction, which is critical to the NRC-to-TGC translation, ensures that each NRC functor can only replace another if they have precisely same signature. Nevertheless, the surface language (SFC) can still have the more general subsumption rules, but signature matching in SFC must have all subsumption coercions made explicit during the translation from SFC to NRC (see discussions on related topics in Section 3.1).

Another interesting aspect is the elaboration of functor application (see Rule 19 in Figure 7). Functor application $f_i(s_i)$ in NRC requires that the argument structure s_i have the syntactically same signature (M_s) as the formal parameter of f_i. This again requires that signature matching at each functor application be made explicit during the SFC-to-NRC translation. The actual application is then done by applying the functor realization r_f of f_i to the realization r_s of s_i. If f_i is a formal functor, that is, r_f is a realization template (A_t), we deduce the result realization through instantiation of f_i's body signature M_s'; if r_f is a realization closure (m_s', R', A_t), we get the result by re-elaborating the functor body (of f_i).

Because NRC does not have datatypes, new stamped types are only generated during signature instantiation (see Figure 8). In fact, only flexible type specifications are assigned new stamped types (see Rule 24). To maintain the mapping from a new stamp to its definitional type path, the instantiation procedure always memoizes the access path (p_x) for the current component.

The static semantics for NRC satisfies a very nice *full-transparency* property: a functor application is assigned the same typing whether we compile it as is or by textually inlining the functor body. Full transparency opens up the possibility of embedding NRC into an F_ω-like calculus such as TGC. After all, F_ω shares a similar property: given a term $e = \Lambda t :: \kappa.e_1$ and a constructor μ of kind κ, type application $e[\mu]$ always has the same type as the result of *inlined* application $[\mu/t]e_1$ where $[\mu/t]$ is a substitution mapping t to μ. The main challenge is then to model functors using type abstraction (Λ) and value abstraction (λ) in TGC.

3.3 Translation from NRC to TGC

The NRC-to-TGC translation uses the same set of deduction rules as in the static semantics (see Figures 7 to 9). There are two key ideas behind our algorithm:

- First, functor application $f_i(s_i)$ in NRC always requires that the argument s_i has exactly the same signature as the formal parameter of f_i. NRC also uses a very restricted set of signature subsumption rules where functors can only match if they have same functor signatures. These restrictions allow us to use the

Relating signature with TGC kind: $\boxed{\vdash^{\kappa} M\_ \Longrightarrow \kappa_t}$

$$\frac{\vdash^{\kappa} D \Longrightarrow lks}{\vdash^{\kappa} \texttt{sig}\ D\ \texttt{end} \Longrightarrow \{lks\}}\ (27) \qquad \vdash^{\kappa} \texttt{type}\ t_i :: \kappa_c = \mu_c \Longrightarrow \varepsilon\ (28) \qquad \vdash^{\kappa} \texttt{type}\ t_i :: \kappa_c \Longrightarrow t_i :: \kappa_c\ (29)$$

$$\frac{\vdash^{\kappa} M_s \Longrightarrow \kappa_t}{\vdash^{\kappa} \texttt{structure}\ s_i :: M_s \Longrightarrow s_i :: \kappa_t}\ (30) \qquad \frac{\vdash^{\kappa} M_f \Longrightarrow \kappa_t}{\vdash^{\kappa} \texttt{functor}\ f_i :: M_f \Longrightarrow f_i :: \kappa_t}\ (31)$$

$$\frac{\vdash^{\kappa} M_s \Longrightarrow \kappa_t \quad \vdash^{\kappa} M_s' \Longrightarrow \kappa_t'}{\vdash^{\kappa} \texttt{fsig}(s_i : M_s) M_s' \Longrightarrow \kappa_t \to \kappa_t'}\ (32) \qquad \vdash^{\kappa} \varepsilon \Longrightarrow \varepsilon\ (33) \qquad \frac{\vdash^{\kappa} D \Longrightarrow lks \quad \vdash^{\kappa} D' \Longrightarrow lks'}{\vdash^{\kappa} DD' \Longrightarrow lks, lks'}\ (34)$$

Relating modtype with TGC type: $\boxed{\vdash^{\tau} (M\_, r\_) \Longrightarrow \mu_t; \sigma_t}$

$$\frac{M_s = \texttt{sig}\ D\ \texttt{end} \qquad R \vdash^{\tau} D \Longrightarrow lcs; lts}{\vdash^{\tau} (M_s, r_s) \Longrightarrow \{lcs\}; \{lts\}}\ (35) \qquad \frac{r_f = (\bullet, \bullet, A_t) \text{ or } (A_t) \quad A_t = (\mu_t, \sigma_t)}{\vdash^{\tau} (M_f, r_f) \Longrightarrow \mu_t; \sigma_t}\ (36)$$

$$R \vdash^{\tau} \varepsilon \Longrightarrow \varepsilon; \varepsilon\ (37) \qquad R \vdash^{\tau} \texttt{type}\ t_i :: \kappa_c = \mu_c \Longrightarrow \varepsilon; \varepsilon\ (39)$$

$$\frac{R \vdash^{\tau} D \Longrightarrow lcs; lts \qquad R \vdash^{\tau} D' \Longrightarrow lcs'; lts'}{R \vdash^{\tau} DD' \Longrightarrow lcs, lcs'; lts, lts'}\ (38) \qquad \frac{\mu_m = R(t_i) \quad \mu_t = \mathbf{m2t}(\mu_m)}{R \vdash^{\tau} \texttt{type}\ t_i :: \kappa_c \Longrightarrow t_i = \mu_t; \varepsilon}\ (40)$$

$$\frac{\vdash^{\tau} (M_s, R(s_i)) \Longrightarrow \mu_t; \sigma_t}{R \vdash^{\tau} \texttt{structure}\ s_i :: M_s \Longrightarrow s_i = \mu_t; s_i : \sigma_t}\ (41) \qquad \frac{\vdash^{\tau} (M_f, R(f_i)) \Longrightarrow \mu_t; \sigma_t}{R \vdash^{\tau} \texttt{functor}\ f_i :: M_f \Longrightarrow f_i = \mu_t; f_i : \sigma_t}\ (42)$$

Figure 9: Relating NRC semantic objects with TGC types

signature to guide our translation.

- Second, an NRC module can be split into a *type* part and a *value* part. We use the signature (e.g., M_s) rather than the modtype to guide splitting. The type (value) part of a structure includes its "type" (value) components plus the type (value) parts of its structure and functor components. By "type" components, we include only those with flexible type specifications in M_s (not those type-abbreviation specs).

  The type part of a functor is a higher-order type function from the type part of its arguments to that of its result; the value part of a functor is a polymorphic function quantified over the type part of its arguments; functor applications can thus be expressed as a combination of type application and value application as in the TGC calculus.

Rules 9–19 in Figure 7 give the translation from the NRC module declarations (expressions) to the TGC declarations (expressions). All module declarations (except type declarations) are translated into the TGC value declarations. Module access path is translated into TGC record selection (Rule 15). We take the liberty of using the same p_s and p_f to denote the TGC selection terms such as $x_i.\cdots.s_i$ and $x_i.\cdots.f_i$. Definitional structure `struct...end` is translated into record construction in TGC (Rule 17).

Each NRC functor, $\texttt{funct}(s_i : M_s)m_b$, is translated into a TGC polymorphic function, $\Lambda s_i :: \kappa_t.\lambda s_i : \sigma_t.e_t$ (see Rule 18). Here, we assume that type and value identifiers in TGC belong to different name space so we can use the same s_i to name both without causing confusion (we can always tell the identifier status in TGC). We use the signature-instantiation procedure in Figure 8 to split functor parameter into two parts: its type part is a type parameter (s_i) of kind κ_t; its value part is a value parameter (s_i) of type σ_t. Kind (κ_t) can be inferred from the signature alone (see Rules 27–34 in Figure 9), and once again only flexible constructors are included in the type part (Rules 28 vs 29).

If we name the type part of the functor parameter as a TGC type constructor "s_i" of kind κ_t, all its components can be assigned a TGC tycon as well. This is again done by the signature instantiation: the μ_t on the left-hand side of Rules 20–26 denote the TGC tycon of the enclosing structure. For example, according to Rule 24, the flexible tycon t_i is translated into a stamped type $n(\kappa_c, (\mu_t.t_i))$ where n is a new stamp, and the selection constructor $\mu_t.t_i$ is the corresponding TGC tycon for the component t_i.

After we instantiate the parameter signature, the functor body m_b is elaborated into a structure realization r_s'. The type part of this functor is expressed as a tycon $\lambda s_i :: \kappa_t.\mu_t$ where μ_t is the type part of m_b. This information is memoized inside the functor realization (r_f) for future use (e.g., Rule 36). The type part of m_b is calculated by the procedure defined in Figure 9: given a structure with signature M_s and realization r_s, its type part is simply a TGC product constructor, counting only those flexible type components (Rule 39 vs. Rule 40). The **m2t** operator in Rule 40 translates a semantic type μ_m into the TGC tycon μ_t by replacing all instances of stamped types $n(\kappa_c', \mu_t')$ with μ_t'.

Translating functor applications is much simpler (Rule 19). A functor application $f_i(s_i)$ is translated into a TGC expression $@(f_i[\mu_t])s_i$. Here, the value part of functor f_i is

signature Z	**type part** (kind κ_Z)	**value part** (type σ_Z)
A : $ASIG$	$\kappa_A = \{t :: \Omega\}$	$\sigma_A = \{\}$
F : $FSIG$	$\kappa_F = \kappa_A \rightarrow \{\}$	$\sigma_F = \forall A_T :: \kappa_A.(\{\} \rightarrow \{\})$
X : $XSIG$	$\kappa_X = \{F :: \kappa_F\}$	$\sigma_X = \{F : \sigma_F\}$
module Z	**type part** (tycon μ_Z)	**value part** (term e_Z)
X	$\mu_X = X_T$	$e_X = X_V$
I	$\mu_I = \{\}$	$e_I = \{\}$
J	$\mu_J = \{t = \texttt{Int}\}$	$e_J = \{\}$
K	$\mu_K = \{\}$	$e_K = @(X_V.F[\mu_J])J$
G	$\mu_G = \lambda X_T :: \kappa_X .$ $\{I = \mu_I, J = \mu_J, K = \mu_K\}$	$e_G = \Lambda X_T :: \kappa_X.\lambda X_V : \sigma_X .$ **let** $I = e_I; J = e_J; K = e_K$ **in** $\{I = I, J = J, K = K\}$

Figure 10: Translating a simple NRC program

translated into a TGC polymorphic function named f_i; the value part of structure s_i is translated into a TGC record named s_i; polymorphic function f_i is applied to the type part of structure s_i, which is a TGC tycon μ_t extracted from s_i's modtype (M_s, r_s).

To prove the correctness of our translation, we need to relate the basis environment in NRC with the kind and type environments in TGC. Given a basis B, we can derive its corresponding TGC environments as follows: the kind environment Δ is just the Δ component maintained inside B; the type environment Γ is calculated by applying the procedure defined in Figure 9, assuming $Rof B \vdash^{\pi} Dof B \Longrightarrow \bullet; lts$, then we convert the list of record fields lts into a TGC type environment in a straightforward manner. In the companion TR [35], we show that as long as the basis B satisfies certain pre-conditions (i.e., it is well-formed and it preserves TGC typing, see TR [35] for detailed definitions), our translation algorithm preserves typing.

Theorem 3.1 (type preservation) *Given a well-formed basis B, suppose B preserves TGC typing, and Δ and Γ are its derived TGC kind and type environments, then*

- *for each NRC module expression $m\_$, if $m\_$ is normalized with respect to B and $B \vdash^{m} m\_ : N ; M\_ ; r\_ \Longrightarrow e_t$ and $\vdash^{\pi} M\_ \Longrightarrow \kappa_t$ and $\vdash^{\pi} (M\_, r\_) \Longrightarrow \mu_t; \sigma_t$, then $\Delta \triangleright \mu_t :: \kappa_t$ and $\Delta; \Gamma \vdash^{t} e_t : \sigma_t$;*
- *for each NRC module declaration d, if d is normalized with respect to B and $B \vdash^{d} d : N ; D ; R \Longrightarrow d_t$, and Γ' is the derived TGC type environment constructed from R and D, then $B \uplus (\emptyset_{\Delta}, N, D, R)$ preserves TGC typing and $\Delta; \Gamma \vdash^{t} d_t : \Gamma'$.*

We conclude this section by applying our algorithm to a sample NRC program. We use ML-like signature declaration to simplify the presentation of NRC signature expressions (keyword `funsig` denotes functor signatures).

```
signature ASIG = sig type t end
funsig FSIG = fsig (A : ASIG) : sig type u = A.t end
signature XSIG = sig functor F : FSIG end
functor G (X : XSIG) =
  struct structure I = struct type t = int end
         structure J = (I : ASIG)
         structure K = X.F(J)
  end.
```

Here, identifiers *ASIG*, *FSIG*, and *XSIG* behave like macros and they are inlined whenever used inside any NRC program. Inside the functor body, structure `I` has signature

```
sig type t = int end,
```

this is not same as the parameter signature of `X.F` so we need to insert an explicit signature matching to create structure `J`. To show how our algorithm works, we give the detailed translation results of every module expression in Figure 10. Here, for every functor parameter Z (e.g., `X` and `A` in the example), we use Z_T to denote its corresponding TGC constructor identifier and Z_V to denote the TGC value identifier; if Z has signature $ZSIG$, we can infer the TGC kind of Z_T (denoted as κ_Z) and the TGC type of Z_V (denoted as σ_Z). Similarly, for every module identifier Z, we give its corresponding TGC type constructor (denoted as μ_Z) and target term expression (denoted as e_Z). Notice the type part for the body of functor `X.F` does not include type `u`. Doing so would force the use of dependent kinds to model κ_X. The fact that type `t` in structure `K` is equivalent to `int` is deduced and propagated by the elaborator.

4 Cross-module program analysis

We can extend the stamp-based semantics for NRC to support cross-module program analysis. Full transparency guarantees that type information be optimally propagated across module boundaries. We could propagate other static information in the same manner, and by doing this, many static program analyses for the core languages can be extended to work across higher-order modules. Because the realization part of a modtype is always hidden inside the compiler, we can freely add new static information into the realization without making any changes to the source-level signature calculus.

To support cross-module inlining, we add a new form of the "type" specifications and declarations into the NRC calculus. We call it *binfo*, meaning the binding information:

$$\begin{array}{llll} spec & D & ::= & ..... \mid \texttt{binfo}\ \ b_i \\ decl & d & ::= & ..... \mid \texttt{binfo}\ \ b_i = e_b \end{array}$$

where e_b is a form of *binfo* expressions, possibly defined as follows:

$$bexp \quad e_b \quad ::= \quad p_b \mid \texttt{Dyn} \mid \texttt{STFun}(e_t) \mid \texttt{STVal}(c) \mid ...$$

Here, `Dyn` denotes a value that we know nothing about at compile time; `STFun` (e) denotes a function that is statically known as a *closed* expression e, written in some typed intermediate language such as TGC; and `STVal` (c) refers to a statically known constant c. Notice we do not make any changes to SFC, instead, the SFC-to-NRC translation can use heuristics (or hints from the programmer) to decide the possible inlining candidates, and then insert the proper *binfo* specifications and declarations into the NRC code.

We extend the semantic objects, NRC realizations, to include a mapping from binfo identifiers to their corresponding binding information. The binfo expressions will be recorded in the realization closure of a functor, just like normal type declarations. All deduction rules remain unchanged, and the binding information will be optimally propagated just as the normal type information.

We then systematically replace each value specification in signatures by a compound structure specification that records all the relevant type, value, and binfo information. Taking the code in Figure 1 as our example, the value specification `val f : s` in `ASIG` is re-interpreted internally as follows:

```
structure f : sig val   dv : s
                  binfo dvB
                  type  tenv
                  type  venvT
                  val   venv : venvT
                  binfo venvB
              end
```

Here, `binfo` v_b is a new form of specification used for binding information. Signature matching on binfo is always transparent. The value component `dv` is `f`'s original definition. The binfo component `dvB` denotes `f`'s binding information. We use `tenv` and `venv` to record all the free type and value identifiers (and paths) in the definition of `f`. We use `venvT` to specify the type of `venv` because each value component must have a type. Finally, the closure `venv` might be a constant itself, so we introduce `venvB` to record its binding information. Given a structure `S` with signature `ASIG`, the access to `S.f` can now be implemented as follows: **(1)** if `dvB` is `Dyn`, or if the optimization is turned off, then `S.f` under the new interpretation is just `S.f.dv`; **(2)** if `dvB` is `STFun` (e), then `S.f` is translated into "@ (e[`S.f.tenv`]) *venv*" where *venv* is c if `venvB` is `STVal` (c), or `S.f.venv` if otherwise; **(3)** if `dvB` is `STVal` (c), then `S.f` is simply c.

Inside each structure body, we replace each value component by a structure declaration that is consistent with our changes on the signatures. For example, function `f` inside structure `SB.F` is replaced by the following:

```
structure f =
  struct fun   dv (x : X.s) = X.f
         binfo dvB = STFun (Λt_e :: Ω.λv_e : t_e.λx : t_e.v_e)
         type  tenv = X.s
         type  venvT = X.s
         val   venv = X.f.dv
         binfo venvB = X.f.dvB
  end
```

If we use the same stamp-based semantics to elaborate the functor application `APP(SB)`, we can deduce that the `f` component in structure `SC` (in Figure 1) is simply "$\lambda x :: int.3$".

Under this algorithm, the binding information of each value component is always propagated *optimally* even across higher-order modules. A more traditional approach would compile the higher-order modules into the usual higher-order functions in the core language, and then perform the heavyweight *control flow analysis* [37] on them. Our module elaboration algorithm is somewhat similar to the abstract execution, but it separates the module-level declarations from the the core-language expressions within. The elaboration is simple and very efficient because the *module-level* code is always small and non-recursive.

5 Implementation

We have implemented both the translation algorithm and the cross-module inlining algorithm in the SML/NJ compiler [36] and the FLINT/ML compiler [34]. The implementation in SML/NJ has been released and in production use since version 109.24 (January 9, 1997). In our implementation, we extended the NRC-to-TGC translation to handle other features in SML'97 [27] such as value components, opaque signature matching, polymorphic types, and recursive datatypes. The translation makes it possible to support type-based optimizations even in the presence of higher-order modules. The cross-module inlining algorithm we implemented currently only inlines and specializes all the primitive functions, but the binding information is fully propagated across the signature matching and functor application. The new inlining algorithm replaced the old ad-hoc algorithm in SML/NJ which does not even propagate inlining information across signature matching. In the following, we briefly explain some of these implementation issues; more details can be found in the companion TR [35].

Adding value components to the NRC calculus is quite trivial. Handling value specifications in signatures requires a utility function (i.e., **c2m**) to convert source-level types (μ_c) into internal semantic types (μ_m). Similarly, to infer the full signature of an arbitrary structure expression, we use a utility function (i.e., **m2c**) to convert the semantic type (μ_m) of each value component back to its source-level counterpart (μ_c)—this is possible because the stamp environment (N) in the basis maintains a one-to-one mapping from each type stamp to its definitional type path.

The semantics given in Section 3.2 always re-elaborates the functor body at each functor application, but much of this is redundant. Under our implementation, only type-related components are re-elaborated at each functor application; all value-related components are type-checked once and for all when the functor definition is processed.

Opaque signature matching [27] can be implemented using the same signature-instantiation algorithm given in Figure 8. All flexible type components are turned into "abstract" types, represented as fresh stamps annotated with its representation type (e.g., $n(\kappa, \mu)$). The NRC-to-TGC translation then converts these stamped types into concrete TGC types by dropping all the stamps. This method does not propagate abstract types into the intermediate language but is sufficient if the underlying compiler does not analyze abstract types.

Although the translation algorithm itself does not improve the efficiency of the module code, it does allow many type-based optimizations to be applied to languages that use ML-style modules. Recent work [36, 38, 33] shows that type-based optimizations dramatically improve the performance of heavily modularized ML programs.

6 Related work

Module systems have been an active research area in the past decade. The ML module system was first proposed by MacQueen [24] and later incorporated into Standard ML [26]. Harper and Mitchell [12] show that the SML'90 module language can be translated into a typed lambda calculus (XML) with dependent types. Together with Moggi, they later show that even in the presence of dependent types, type-checking of XML is still decidable [13], thanks to the phase-distinction property of ML-style modules. The SML'90 module language, however, contains several major problems; for example, type abbreviations are not allowed in signatures, opaque signature matching is not supported, and modules are first-order only. These problems were heavily researched [11, 19, 20, 23, 39, 25, 16] and mostly resolved in SML'97 [27]. The main remaining issue is with the design of higher-order modules, with proposals ranging from fully transparent ones [25], to applicative functors [20, 7], or abstract functors [11, 19, 23]. Fully transparent modules are most expressive, but it is not clear whether they are absolutely necessary; they also interact poorly with *true separate compilation* [19]. This paper shows that at least from the implementation point of view, full transparency is important in providing optimal support to efficient cross-module compilation.

The question of whether higher-order modules can be compiled into simple F_ω-like calculus has been open for a while. Several recent papers [13, 4, 20] have attacked variants of this problem with different motivations; however, they all impose severe restrictions to their module languages. The algorithm hidden inside Harper, Mitchell, and Moggi's phase-distinction paper [13] is most related, however, it does not support type abbreviation and sharing in signatures.[2] Supporting type abbreviation is non-trivial, as discovered by Morrisett [30] and also demonstrated in this paper. Harper and Stone [15] give a new type-theoretical semantics for the entire SML'97, however, their internal language, IL, contains a separate module calculus that uses translucent signatures. Biswas [4] gives a semantics for the MacQueen-Tofte modules based on simple polymorphic types; however, his algorithm does not support parameterized type constructors. Another difference is that in his scheme, functors are not considered as higher-order type constructors, instead, he has to encode certain type constructors of kind Ω using higher-order types; this significantly complicates his semantics. Finally, Leroy [20] uses applicative functors to achieve full transparency, but his approach handles limited functor arguments only; Courant [7]'s semantics does not have such restriction, but he did not give a translation of his calculus into the F_ω calculus.

Both Lillibridge [23] and Leroy [21] discussed how to add value identities to their module interfaces though neither of them gave any actual algorithm. Blume and Appel [5] proposed a cross-module inlining algorithm that supports inlining of functions with free variables. Their algorithm is carried in an untyped setting, so type specialization is not directly supported; neither does their algorithm guarantee the fully transparent propagation of the inlining information. Our algorithm does guarantee the optimal propagation, but at the price of further complicating the module elaboration. We are currently working together on an inlining algorithm that combines the best of both schemes.

Type-directed compilation has received much attention lately, but little has been done to extend it to work across higher-order modules. Shao and Appel [36] extended Leroy's representation analysis [18] to work for the SML'90 modules; their algorithm works only for the pure-coercion-based representation analysis [18]. The algorithm in this paper translates the module language into the F_ω calculus, so type-based optimizations [18, 33, 14, 28, 40] that work for F_ω immediately work for higher-order modules as well.

[2] Although the paper by Harper *et al* [13] was published in 1990, the importance of its phase-splitting algorithm was not recognized until very recently. In fact, we reinvented the same algorithm while working on the type-directed compilation of ML-style modules during 1996. The first version of our algorithm was presented at the IFIP WG2.8 meeting in September 1996. It was at that meeting when Bob Harper pointed us to the phase-distinction paper. In January 1997, Greg Morrisett [30] told us that he had problems adapting the original phase-splitting algorithm [13] to SML [27] since it does not support type abbreviations in signatures. As a result, the TIL compiler today is still using an intermediate language with dependent "singleton" kinds rather than the plain F_ω calculus.

7 Conclusions

We have presented a series of techniques for compiling across higher-order modules. These techniques have been implemented and released with the SML/NJ compiler since version 109.24 (January 9, 1997). The main contribution of our work is the translation algorithm from ML-style modules (SML'97 extended with MacQueen-Tofte higher-order modules) to the F_ω calculus. Without such translation, none of those important type-based optimizations [14, 33, 34] would apply to the full SML language. We have also presented ways to extend various program analyses to work across higher-order modules; in fact, we show that for fully transparent modules, static information can always be optimally propagated across the module boundaries. Finally, we have presented a new and more complete formal definition for the MacQueen-Tofte higher-order modules; our new semantics covers a much richer language and solves all the remaining technical problems in MacQueen and Tofte's original proposal [25].

Availability

The implementation discussed in this paper is now released with the Standard ML of New Jersey (SML/NJ) compiler and the FLINT/ML compiler [34]. SML/NJ is a joint work by Lucent, Princeton, Yale and AT&T. FLINT is a modern compiler infrastructure developed at Yale University. Both FLINT and SML/NJ are available from the following web site:

`http://flint.cs.yale.edu`

Acknowledgement

I would like to thank Andrew Appel, Christopher League, David MacQueen, Bratin Saha, Chris Stone, Valery Trifonov, and the anonymous referees for their comments and suggestions on an early version of this paper. The implementation of higher-order modules inside SML/NJ is joint work with David MacQueen at Lucent Technologies.

References

[1] A. Aiken and N. Heintze. Constraint-based program analysis. POPL'95 Tutorial, January 1995.

[2] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] E. Biagioni, R. Harper, P. Lee, and B. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *1994 ACM Conference on Lisp and Functional Programming*, pages 55–64, New York, June 1994. ACM Press.

[4] S. K. Biswas. Higher-order functors with transparent signatures. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 154–163, New York, Jan 1995. ACM Press.

[5] M. Blume and A. W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 112–124. ACM Press, June 1997.

[6] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, Stanford, California, March 1992.

[7] J. Courant. An applicative module calculus. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development: LNCS Vol 1214*, pages 622–636, New York, 1997. Springer-Verlag.

[8] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Ninth Annual ACM Symp. on Principles of Prog. Languages*, pages 207–212, New York, Jan 1982. ACM Press.

[9] L. George. MLRISC: Customizable and reusable code generators. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, 1997.

[10] J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur.* PhD thesis, University of Paris VII, 1972.

[11] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-first Annual ACM Symp. on Principles of Prog. Languages*, pages 123–137, New York, Jan 1994. ACM Press.

[12] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. on Programming Languages and Systems*, 15(2):211–252, April 1993.

[13] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 341–344, New York, Jan 1990. ACM Press.

[14] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[15] R. Harper and C. Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1997.

[16] M. P. Jones. Using parameterized signatures to express modular structure. In *Twenty-third Annual ACM Symp. on Principles of Prog. Languages*, pages 68–78, New York, Jan 1996. ACM Press.

[17] N. D. Jones. Partial evaluation. POPL'91, tutorial handout, January 1991.

[18] X. Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, Jan 1992. ACM Press. Longer version available as INRIA Tech Report.

[19] X. Leroy. Manifest types, modules, and separate compilation. In *Twenty-first Annual ACM Symp. on Principles of Prog. Languages*, pages 109–122, New York, Jan 1994. ACM Press.

[20] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 142–153, New York, Jan 1995. ACM Press.

[21] X. Leroy. A modular module system. Technical report 2866, INRIA, April 1996.

[22] X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):1–32, September 1996.

[23] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997. Tech Report CMU-CS-97-122.

[24] D. MacQueen. Using dependent types to express modular structure. In *Proc. 13th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 277–286. ACM Press, 1986.

[25] D. MacQueen and M. Tofte. A semantics for higher order functors. In *The 5th European Symposium on Programming*, pages 409–423, Berlin, April 1994. Spinger-Verlag.

[26] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

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

[28] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. 23rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[29] G. Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. Tech Report CMU-CS-95-226.

[30] G. Morrisett. Personal Communication, Cornell University, January 1997.

[31] G. Nelson, editor. *Systems programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[32] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

[33] Z. Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 85–98. ACM Press, June 1997.

[34] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[35] Z. Shao. Typed cross-module compilation. Technical Report YALEU/DCS/RR-1126, Department of Computer Science, Yale University, New Haven, CT, June 1998.

[36] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN '95 Conf. on Prog. Lang. Design and Implementation*, pages 116–129. ACM Press, 1995.

[37] O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon Univ., Pittsburgh, Pennsylvania, May 1991. CMU-CS-91-145.

[38] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 181–192. ACM Press, 1996.

[39] M. Tofte. Principal signatures for high-order ML functors. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 189–199, New York, Jan 1992. ACM Press.

[40] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 1–11, New York, June 1994. ACM Press.