Yale University
Department of Computer Science

Typed Cross-Module Compilation

Zhong Shao
Yale University

YALEU/DCS/TR-1126
June 27, 1998

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.

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.

Typed Cross-Module Compilation

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

Technical Report YALEU/DCS/TR-1126
June 27, 1998

1 Introduction

Modular programming has proven to be extremely valuable in the development and maintenance of large software systems [BHLM94, Nel91, Geo97]. Many modern programming languages such as Modula-3 [Nel91] and Standard ML [MTH90, MTHM97] 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 [MT94, HL94, Ler95] are widely recognized as one of the most powerful module constructs in existence today. Recent work on the type-theoretic foundations of ML modules [HL94, Ler94, MTHM97] has cleaned up many rough spots in the original design [MTH90]. Still, the semantics for higher-order modules involves the use of dependent types [Mac86, HM93] or translucent signatures [HL94, Ler94]. MacQueen and Tofte [MT94] have shown that even a small restriction on signature matching [Ler94, HL94] 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 [DM82].

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 [Ler92, Sha97a], type

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.

```
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

specialization [Cha92], intensional type analysis [HM95], typed closure conversion [MMH96], tagless garbage collection [Tol94], to name just a few. All of these are performed on the variants of core ML [DM82] or the polymorphic λ-calculus F_ω [Gir72, Rey74]. 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. 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 [Ler92, Sha97a] and intensional type analysis [HM95] 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 [App92], partial evaluation [Jon91],

and constraint-based analysis [AH95], 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 [HMM90] 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 [MT94] 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 [HMM90]: 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 [Ler92, Tol94, HM95, MMH96, Sha97a] 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 [HMM90, Bis95, Ler95] 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 [HMM90], no parameterized types [Bis95], and limited forms of functor arguments [Ler95]).
- Our compilation algorithm can handle the entire SML'97 language [MTHM97] 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.

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

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

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

Figure 3: Static semantics for TGC: a summary

- 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 [MT94] 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 [HMM90] 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 [HM95], 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, as summarized in Figure 4, consists of a collection of rules for constructor formation, constructor equivalence, type formation, type equivalence, term formation, and declaration forma-

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

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

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

Figure 4: Selected typing rules for TGC

tion. 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 $\Delta; \Gamma \vdash e_t : \sigma_t$, meaning that term e_t is assigned type σ_t under kind environment Δ and type environment Γ. Figure 4 gives the typing rules for the product-related constructors and terms. The rest of the rules are all standard, as shown in Appendix A. 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 [Mor95, HM95].

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[MTHM97] extended with the MacQueen-Tofte higher-order modules [MT94]. 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. Appendix B 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 5. 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

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 5: Syntax of the normalized module calculus NRC

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 [Ler96b]. 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 Appendix B 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.
```

Stamp	n	$\equiv$	same as i	*Path*	p_x	$::=$	$p_s \mid \varepsilon$
Tycon	μ_m	$::=$	$n(\kappa_c, \mu_t) \mid \alpha \mid int \mid \mu_m \to \mu'_m$	*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 6: Semantic objects for NRC

signature subsumption	$B \vdash^{s} M_s \leq M'_s$	Figure 8
module declaration	$B \vdash^{d} d : N ; D ; R \Longrightarrow d_t$	Figure 9
module expression	$B \vdash^{m} m\_ : N ; M\_ ; r\_ \Longrightarrow e_t$	Figure 9
signature instantiation	$\mu_t ; p_x ; B \vdash^{I} M_s : N ; r_s \Longrightarrow \sigma_t$	Figure 10
signature kind translation	$\vdash^{\kappa} M\_ \Longrightarrow \kappa_t$	Figure 11
module type translation	$\vdash^{\kappa} (M\_, r\_) \Longrightarrow \mu_t ; \sigma_t$	Figure 11

Figure 7: NRC semantics and translation: a summary

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 Appendix B for details).

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 [MT94] 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 6 defines the semantic objects used by our new semantics. Figure 7 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 $\emptyset_{\mathrm{N}}$, $\emptyset_{\mathrm{R}}$, 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

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

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

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

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

Figure 8: Signature subsumption in NRC

all of its functor definitions have distinctly named formal parameters; these conditions are enforced by the SRC-to-NRC translation.

Returning to Figure 6, 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,

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 9: Module semantics and its translation into TGC

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

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

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

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

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

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

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

Figure 10: Signature instantiation and its translation into TGC

and vice versa. 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 8): they are much more restrictive than those used by MacQueen and Tofte [MT94]. 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 9). 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),

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

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

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

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

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

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

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

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

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

Figure 11: Relating NRC semantic objects with TGC types

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 10). 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.

To reason about the static semantics, we first define a set of functions and notations used in the formal development. Function *freeStamps* returns the set of free stamps occurred inside a semantic object. Because stamps only occur free inside μ_m, the free stamps in a realization can be calculated as follows:

$$\begin{array}{lcl} \mathit{freeStamps}(r_t) & = & \mathit{freeStamps}(\mu_m) \ \text{ if } \ r_t = \mu_m; \\ \mathit{freeStamps}(r_s) & = & \cup_{r \in R}(\mathit{freeStamps}(r)) \ \text{ assuming } \ r_s = R; \\ \mathit{freeStamps}(r_f) & = & \mathit{freeStamps}(R \text{of} B) \ \text{ if } \ r_f = (\bullet, B, \bullet); \ \text{ or } \ \emptyset_{\mathrm{N}} \ \text{ if } \ r_f = (A_t). \end{array}$$

Function *defPaths* returns the set of access paths defined by a specification environment (or a realization

environment or an NRC module term). For example, the access paths defined by a realization environment can be calculated as follows:

$$\mathit{defPaths}(r) \quad = \quad \cup_{x \in \mathrm{Dom}(R)}(\{x\} \cup \{x.p \mid p \in \mathit{defPaths}(R(x))\} \text{ if } r = R; \text{ or } \emptyset_{\mathrm{p}} \text{ otherwise.}$$

Function *freePaths* returns the set of access paths occurred free in a specification environment (or a signature or a module term), taking consideration of removing those locally defined paths. The *freePaths* operation is always closed under the prefix operation. Finally, we say that two semantic objects are syntactically equivalent if they match precisely according to the definition in Figure 5 and 9; we then define that two realization environments R and R' are *consistent* with each other if for each $p \in (\mathit{defPaths}(R) \cap \mathit{defPaths}(R'))$, realization $R(p)$ and $R'(p)$ are syntactically equivalent; similarly, two specification environments D and D' are *consistent* if for each $p \in (\mathit{defPaths}(D) \cap \mathit{defPaths}(D'))$, specification $D(p)$ and $D'(p)$ are syntactically equivalent.

Definition 3.1 (normalized programs and environments)

- *An NRC program fragment X (e.g., expression, declaration) is normalized if there are no duplicate bindings within each scope, and all of its functor definitions have distinctly named formal parameters.*
- *An NRC program fragment X (e.g., signature, expression, declaration) is normalized with respect to basis $B = (\Delta, N, D, R)$ if X is normalized and it does not rebind any functor parameter names in (D, R), nor does it redefine any identifiers in* $\mathrm{Dom}(R)$.
- *An environment (D, R) is normalized if (1) for each x in D, $D(x)$ is normalized, and (2) given any two functor paths p_f and p'_f in $\mathit{defPaths}(D)$ and suppose $M_f = D(p_f)$ and $M'_f = D(p'_f)$ then either $(M_f, R(p_f))$ and $(M'_f, R(p'_f))$ are syntactically equivalent, or all functor signatures in M_f and M'_f have different parameter names.*

Definition 3.2 (well-formed semantic objects)

- *A basis B is well-formed if its modtype environment $(D\mathrm{of}B, R\mathrm{of}B)$ is well-formed with respect to B.*
- *A modtype environment (D', R') is well-formed with respect to basis $B = (\Delta, N, D, R)$ if (1) (D', R') is normalized, and D' is consistent with D and R' is consistent with R; (2) $\mathit{freeStamps}(R') \subseteq \mathrm{Dom}(N)$, and for each $n \in \mathit{freeStamps}(R')$, if $N(n) = p_t$ then $p_t \in \mathit{defPaths}(R')$; (3) $\mathit{defPaths}(D') \subseteq \mathit{defPaths}(R')$, and for each type component $p_t \in \mathit{defPaths}(D')$, p_t is well-formed inside (D', R'); for each module component $p\_ \in \mathit{defPaths}(D')$, the modtype $(D'(p\_), R'(p\_))$ is well-formed with respect to B.*
- *A type component p_t inside environment (D, R) is well-formed if suppose $D(p_t)$ is a type specification with kind κ_c and (optional) type definition μ_c, and $R(p_t)$ is μ_m, then μ_m has kind κ_c, and if μ'_m is the result of replacing each free type path p'_t in μ_c with $R(p'_t)$, then $\mu_m \equiv_m \mu'_m$.*
- *A structure modtype (M_s, r_s) is well-formed with respect to basis B if suppose $M_s =$ `sig` D' `end` and $r_s = R'$, then the environment (D', R') is well-formed with respect to B.*
- *A functor modtype (M_f, r_f) is well-formed with respect to basis $B = (\Delta, N, D, R)$ if assuming $r_f = (A_t)$, then $\mathit{freePaths}(M_f) \subseteq \mathit{defPaths}(R)$; or assuming $r_f = (m_b, B', A_t)$ where $B' = (\bullet, \bullet, D', R')$, let $M_f =$ `fsig`$(s_i : M_s)M'_s$ and $m_f =$ `funct` $(s_i : M_s)m_b$, then m_f is normalized with respect to B', and $\mathit{freePaths}(m_f) \subseteq \mathit{defPaths}(R')$, and (D', R') is well-formed with respect to B.*

Lemma 3.3 *Given a well-formed basis $B = (\Delta, N, D, R)$, an NRC signature M_s, a TGC constructor μ_t, if M_s is normalized with respect to B and $\mu_t ; \varepsilon ; B \vdash^{\Gamma} M_s : N' ; r_s \Longrightarrow \bullet$, then $\mathrm{Dom}(N) \cap \mathrm{Dom}(N') = \emptyset$, and $freePaths(M_s) \subseteq defPaths(R)$, and modtype (M_s, r_s) is well-formed with respect to $B' = (\Delta, N \uplus N', D, R)$.*

Proof: By structural induction on the signature expression M_s. The side condition $\mu_m = \mathbf{c2m}(B, \mu_c)$ in Rule 23 guarantees that all free type paths in M_s is defined in R. □

Lemma 3.4 *Given a well-formed basis $B = (\Delta, N, D, R)$, suppose d is an NRC declaration, if d is normalized with respect to B and $B \vdash^{d} d : N' ; D' ; R' \Longrightarrow \bullet$, then $\mathrm{Dom}(N') \cap \mathrm{Dom}(N) = \emptyset$, and $\mathrm{Dom}(D') \cap \mathrm{Dom}(D) = \emptyset$, and $\mathrm{Dom}(R') \cap \mathrm{Dom}(R) = \emptyset$, and $freePaths(d) \subseteq defPaths(R)$, and the basis $(\Delta, N \uplus N', D \uplus D', R \uplus R')$ is well-formed as well. Similarly, suppose $m\_$ is an NRC expression, if $m\_$ is normalized with respect to B and $B \vdash^{m} m\_ : N' ; M'\_ ; r'\_ \Longrightarrow \bullet$, then $\mathrm{Dom}(N') \cap \mathrm{Dom}(N) = \emptyset$, and $freePaths(m\_) \subseteq defPaths(R)$, and $(M'\_, r'\_)$ is well-formed with respect to basis $(\Delta, N \uplus N', D, R)$.*

Proof: Using Lemma 3.3 and by induction on derivation trees [Win93, Chapter 3]. □

The first property we can show is that our static semantics always produce typings that are unique up to the α-renaming of type stamps. More specifically, we say two stamp environments N and N' are α-equivalent if there exists a one-to-one mapping φ from $\mathrm{Dom}(N)$ to $\mathrm{Dom}(N')$ such that for each $n \in \mathrm{Dom}(N)$, $N(n)$ is syntactically equivalent to $N'(\varphi(n))$. We also say that two basis $B = (\Delta, N, D, R)$ and $B' = (\Delta', N', D', R')$ are α-equivalent if (1) N and N' are α-equivalent under the mapping φ; (2) $R' = \varphi(R)$; and (3) D and D' are syntactically equivalent.

Lemma 3.5 *Given two well-formed basis B and B', an NRC signature M_s, a TGC constructor μ_t, and a path p_x, suppose B and B' are α-equivalent, and M_s is normalized with respect to B and B', then if both $\mu_t ; p_x ; B \vdash^{\Gamma} M_s : N ; r_s \Longrightarrow \bullet$ and $\mu_t ; p_x ; B' \vdash^{\Gamma} M_s : N' ; r'_s \Longrightarrow \bullet$ are true, then $N \mathrm{of} B \uplus N$ and $N \mathrm{of} B' \uplus N'$ are α-equivalent under a mapping φ, with $r_s = \varphi(r'_s)$.*

Proof: By structural induction on signature expression. □

Theorem 3.6 (unique typing) *Suppose B and B' are two well-formed basis, $m\_$ is an NRC module expression, d is an NRC module declaration, suppose B and B' are α-equivalent, both $m\_$ and d are normalized with respect to B and B', then (1) if both $B \vdash^{m} m\_ : N ; M\_ ; r\_ \Longrightarrow \bullet$ and $B' \vdash^{m} m\_ : N' ; M'\_ ; r'\_ \Longrightarrow \bullet$ are true, then $M\_$ and $M'\_$ are syntactically equivalent, and $N \mathrm{of} B \uplus N$ and $N \mathrm{of} B' \uplus N'$ are α-equivalent under a mapping φ, with $r\_ = \varphi(r'\_)$; (2) if both $B \vdash^{d} d : N ; D ; R \Longrightarrow \bullet$ and $B' \vdash^{d} d : N' ; D' ; R' \Longrightarrow \bullet$ are true, then $B \uplus (\emptyset_{\Delta}, N, D, R)$ and $B' \uplus (\emptyset_{\Delta}, N', D', R')$ are α-equivalent.*

Proof: By induction on derivation trees and Lemma 3.5. □

We can also show that under our static semantics, elaboration of NRC programs always terminates.

Theorem 3.7 (termination) *Suppose B is a well-formed basis and d is an NRC declaration, if d is normalized with respect to B, then elaboration $B \vdash^{d} d : N ; D ; R \Longrightarrow \bullet$ always terminates. Similar proposition holds for module expressions.*

Proof: We relate the derivation tree of our static semantics with the derivation tree in an environment-based operational semantics for a simply typed λ-calculus (extended with labelled records and integers). Each functor realization in our semantics relates to a function closure that calculates the total number of functor applications

performed when the functor is applied. An infinite derivation tree under our semantics would then correspond to an infinite derivation for evaluating a simply typed λ-expression. This contradicts with the fact that simply typed λ-calculus is strongly normalized. □

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 9 to 11). 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 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 9 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 10 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 11), 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 11: 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 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 11, assuming $R\text{of}B \vdash^* D\text{of}B \Longrightarrow \bullet; lts$, then we convert the list of record fields lts into a TGC type environment in a straightforward manner. In the following, we state the type preservation theorem and sketch its proof.

Definition 3.8 *A well-formed basis $B = (\Delta, N, D, R)$ preserves TGC typing if*

- *for each type component $p_t \in defPaths(D)$, suppose $D(p_t)$ has kind κ_t, and $\mu_t = \mathbf{m2t}(R(p_t))$, then $\Delta \triangleright \mu_t :: \kappa_t$;*
- *for each structure component $p_s \in defPaths(D)$, suppose $M_s = D(p_s)$ and $r_s = R(p_s)$ and $\vdash^* M_s \Longrightarrow \kappa_t$ and $\vdash^* (M_s, r_s) \Longrightarrow \mu_t; \sigma_t$, then $\Delta \triangleright \mu_t :: \kappa_t$ and $\Delta \triangleright \sigma_t$;*
- *for each functor component $p_f \in defPaths(D)$, suppose $M_f = D(p_f)$ and $r_f = R(p_f)$ and $\vdash^* M_f \Longrightarrow \kappa_t$ and $\vdash^* (M_f, r_f) \Longrightarrow \mu_t; \sigma_t$, then there exists TGC entities κ'_t, κ''_t, μ'_t, σ'_t, and σ''_t such that $\kappa_t = (\kappa'_t \to \kappa''_t)$ and $\Delta \triangleright \mu_t \equiv_t (\lambda s_i :: \kappa'_t.\mu'_t) :: \kappa_t$ and $\Delta \triangleright \sigma_t \equiv_t (\forall s_i :: \kappa'_t.\sigma'_t \to \sigma''_t)$; furthermore, if $r_f = (m'_b, B', \bullet)$, then B' preserves TGC typing as well.*

Definition 3.9 *Given two well-formed basis $B = (\Delta, N, D, R)$ and $B' = (\Delta', N', D', R')$, we say that B is related to B' via a TGC substitution ρ (mapping TGC type variables to TGC type constructors) if*

- *both B and B' preserve TGC typings;*
- *$\mathrm{Dom}(\rho) \subseteq \mathrm{Dom}(\Delta)$ and $\mathrm{Dom}(\rho) \cap \mathrm{Dom}(\Delta') = \emptyset$, and for each x in $\mathrm{Dom}(\rho)$, suppose $\Delta(x) = \kappa_t$ then $\Delta' \triangleright \rho(x) :: \kappa_t$;*
- *for each type component $p_t \in (defPaths(D) \cap defPaths(D'))$, both $D(p_t)$ and $D'(p_t)$ have the same kind κ_t; furthermore, let $\mu_t = \mathbf{m2t}(R(p_t))$ and $\mu'_t = \mathbf{m2t}(R'(p_t))$, then $\Delta' \triangleright \rho(\mu_t) \equiv_t \mu'_t :: \kappa_t$;*
- *for each module component $p\_ \in (defPaths(D) \cap defPaths(D'))$, both $D(p\_)$ and $D'(p\_)$ have the same signature $M\_$; furthermore, if $\vdash^* M\_ \Longrightarrow \kappa_t$ and $\vdash^* (M\_, R(p\_)) \Longrightarrow \mu_t; \sigma_t$ and $\vdash^* (M\_, R'(p\_)) \Longrightarrow \mu'_t; \sigma'_t$, then $\Delta' \triangleright \rho(\mu_t) \equiv_t \mu'_t :: \kappa_t$ and $\Delta' \triangleright \rho(\sigma_t) \equiv_t \sigma'_t$.*

Lemma 3.10 *Given a well-formed basis* $B = (\Delta, N, D, R)$*, a TGC type constructor* μ_t*, a path* p_x*, and a signature* M_s*, suppose* B *preserves TGC typing, also suppose* $\vdash^{\omega} M_s \Longrightarrow \kappa_t$ *and* $\Delta \triangleright \mu_t :: \kappa_t$ *and* $\mu_t ; p_x ; B \vdash^{\Gamma} M_s : N ; r_s \Longrightarrow \sigma_t$ *and* $\vdash^{\omega} (M_s, r_s) \Longrightarrow \mu'_t ; \sigma'_t$*, then* $\Delta \triangleright \mu_t \equiv_t \mu'_t :: \kappa_t$ *and* $\Delta \triangleright \sigma_t \equiv_t \sigma'_t$*.*

Proof: By structural induction on the signature expression M_s. □

Lemma 3.11 *Given two well-formed basis* B *and* B'*, an NRC signature expression* M_s*, two TGC type constructors* μ_t *and* μ'_t*, and an NRC path* p_x*, suppose* B *is related to* B' *via a TGC substitution* ρ*, also suppose* $\vdash^{\omega} M_s \Longrightarrow \kappa_t$ *and* $\Delta \mathrm{of} B' \triangleright \rho(\mu_t) \equiv_t \mu'_t :: \kappa_t$ *and* $\mu_t ; p_x ; B \vdash^{\Gamma} M_s : N ; r_s \Longrightarrow \sigma_t$ *and* $\mu'_t ; p_x ; B' \vdash^{\Gamma} M_s : N' ; r'_s \Longrightarrow \sigma'_t$*, then* $\Delta \mathrm{of} B' \triangleright \rho(\sigma_t) \equiv_t \sigma'_t$*.*

Proof: By structural induction on the signature expression M_s. The key insight is that the TGC type (σ_t) of a modtype can be completely determined by its signature plus its corresponding TGC tycon (μ_t). □

Lemma 3.12 *Given a well-formed basis* $B = (\Delta, N, D, R)$ *and an NRC path* p_x*, suppose* B *preserves TGC typing, then for each structure component* p_s *in* $defPaths(D)$*, assume* $D(p_s) = M_s$ *and* $R(p_s) = r_s$ *and* $\vdash^{\omega} M_s \Longrightarrow \kappa_t$ *and* $\vdash^{\omega} (M_s, r_s) \Longrightarrow \mu_t ; \sigma_t$ *, let* s_i *be a TGC type variable not in* $\mathrm{Dom}(\Delta)$ *and* $\Delta' = \{s_i :: \kappa_t\}$*, also let* $B' = B \uplus (\Delta', \emptyset_{\mathrm{N}}, \emptyset_{\mathrm{D}}, \emptyset_{\mathrm{R}})$ *, then if* $s_i ; p_x ; B' \vdash^{\Gamma} M_s : N' ; r'_s \Longrightarrow \sigma'_t$ *and* $\mu_t ; p_x ; B \vdash^{\Gamma} M_s : N'' ; r''_s \Longrightarrow \sigma''_t$ *, we have* $\Delta \mathrm{of} B \triangleright ([\mu_t / s_i](\sigma'_t)) \equiv_t \sigma''_t$*.*

Proof: A straightforward application of Lemma 3.11. □

Lemma 3.13 *Given well-formed basis* B *and* B'*, if* B *is related to* B' *via a TGC substitution* ρ*, then*

- *for each NRC module expression* $m\_$*, if* $m\_$ *is normalized with respect to* B *and* B'*, and both* $B \vdash^{d} m\_ : N ; M\_ ; r\_ \Longrightarrow \bullet$ *and* $B' \vdash^{m} m\_ : N' ; M'\_ ; r'\_ \Longrightarrow \bullet$ *are true, then* $M\_$ *and* $M'\_$ *are syntactically equivalent; furthermore, if* $\vdash^{\omega} M\_ \Longrightarrow \kappa_t$ *and* $\vdash^{\omega} (M\_, r\_) \Longrightarrow \mu_t ; \sigma_t$ *and* $\vdash^{\omega} (M'\_, r'\_) \Longrightarrow \mu'_t ; \sigma'_t$*, then* $\Delta \mathrm{of} B' \triangleright \rho(\mu_t) \equiv_t \mu'_t :: \kappa_t$ *and* $\Delta \mathrm{of} B' \triangleright \rho(\sigma_t) \equiv_t \sigma'_t$*.*
- *for each NRC module declaration* d*, if* d *is normalized with respect to* B *and* B'*, and both* $B \vdash^{d} d : N ; D ; R \Longrightarrow \bullet$ *and* $B' \vdash^{d} d : N' ; D' ; R' \Longrightarrow \bullet$ *are true, then* D *and* D' *are syntactically equivalent; furthermore, basis* $B \uplus (\emptyset_{\Delta}, N, D, R)$ *is related to* $B \uplus (\emptyset_{\Delta}, N', D', R')$ *via the TGC substitution* ρ *as well.*

Proof: By induction on the derivations and using Lemma 3.10 to 3.12. □

Theorem 3.14 (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* $\vdash^{\omega} M\_ \Longrightarrow \kappa_t$ *and* $\vdash^{\omega} (M\_, r\_) \Longrightarrow \mu_t ; \sigma_t$*, then* $\Delta \triangleright \mu_t :: \kappa_t$ *and* $\Delta ; \Gamma \vdash^{\Gamma} e_t : \sigma_t$*;*
- *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* $\Delta ; \Gamma \vdash^{\Gamma} d_t : \Gamma'$*.*

Proof: By induction on the derivations. The only nontrivial case is on derivations ended with Rule 19. To deduce $B \vdash^m f_i(s_i) : \bullet ; M'_s ; r'_s \Longrightarrow @(f_i[\mu_t])s_i$, we must already have $B(f_i) = (M_f, r_f)$ and $B(s_i) = (M_s, r_s)$ and $M_f = \texttt{fsig}(s'_i : M_s)M'_s$ and $D' = \{\texttt{structure } s'_i : M_s\}$ and $R' = \{s'_i \mapsto r_s\}$ and $\vdash^{\infty} M_s \Longrightarrow \kappa_t$ and $\vdash^{\infty} M'_s \Longrightarrow \kappa'_t$ and $\vdash^{\infty} (M_s, r_s) \Longrightarrow \mu_t; \sigma_t$. Assuming $\vdash^{\infty} (M'_s, r'_s) \Longrightarrow \mu'_t; \sigma'_t$, then we need to prove both $\Delta \triangleright \mu'_t :: \kappa'_t$ and $\Delta; \Gamma \vdash @(f_i[\mu_t])s_i : \sigma'_t$.

In the case of $r_f = (A_t)$, the auxiliary TGC information A_t must be of form $(\lambda s'_i :: \kappa_t . \mu''_t,\ \forall s'_i :: \kappa_t . \sigma''_t \rightarrow \sigma'''_t)$; furthermore, such functor realization must be introduced by Rule 26 in Figure 10. Thus if we let $\Delta' = \{s'_i :: \kappa_t\}$, we should also have $s'_i ; s'_i ; B \uplus (\Delta', \emptyset_{\text{N}}, \emptyset_{\text{D}}, \emptyset_{\text{R}}) \vdash^{\prime} M_s : \bullet ; r''_s \Longrightarrow \sigma''_t$. From Lemma 3.10, we have $\vdash^{\infty} (M_s, r''_s) \Longrightarrow s'_i; \sigma''_t$; because $\vdash^{\infty} (M_s, r'_s) \Longrightarrow \mu_t; \sigma_t$, from Lemma 3.12, we can deduce $\Delta \triangleright ([\mu_t / s'_i](\sigma''_t)) \equiv_t \sigma_t$. From the premise of Rule 19, if we name $B'' = B \uplus (\emptyset_{\Delta}, \emptyset_{\text{N}}, D', R')$, we have $[\mu_t / s'_i](\mu''_t) ; \varepsilon ; B'' \vdash^{\prime} M'_s : \bullet ; r'_s \Longrightarrow \sigma'_t$. But from Rule 26, if we let $R'' = \{s'_i \mapsto r''_s\}$ and $B''' = B \uplus (\Delta', \emptyset_{\text{N}}, D', R'')$, we also have $\mu''_t ; \varepsilon ; B''' \vdash^{\prime} M'_s : \bullet ; \bullet \Longrightarrow \sigma'''_t$. Because $\vdash^{\infty} (M'_s, r'_s) \Longrightarrow \mu'_t; \sigma'_t$, so from Lemma 3.10, we have $\Delta \triangleright [\mu_t / s'_i]\mu''_t \equiv_t \mu'_t :: \kappa'_t$ which also implies $\Delta \triangleright \mu'_t :: \kappa'_t$. At the same time, we can see B''' is related to B'' via a TGC substitution $[\mu_t / s'_i]$, thus from Lemma 3.11, we can deduce $\Delta \triangleright [\mu_t / s'_i]\sigma'''_t \equiv_t \sigma'$; thus $\Delta \triangleright [\mu_t / s'_i](\sigma''_t \rightarrow \sigma'''_t) \equiv_t \sigma_t \rightarrow \sigma'$, because f_i has type $\forall s'_i :: \kappa_t . \sigma''_t \rightarrow \sigma'''_t)$, and s_i has type σ_t, so $\Delta; \Gamma \vdash @(f_i[\mu_t])s_i : \sigma'_t$.

In the case of $r_f = (m'_b, B', A_t)$, the auxiliary information A_t must be of form $(\lambda s'_i :: \kappa_t . \mu''_t,\ \forall s'_i :: \kappa_t . \sigma''_t \rightarrow \sigma'''_t)$; furthermore, such functor realization must be introduced by Rule 18 in Figure 9; so if we let $\Delta' = \{s'_i :: \kappa_t\}$ and $s'_i ; s'_i ; B \uplus (\Delta', \emptyset_{\text{N}}, \emptyset_{\text{D}}, \emptyset_{\text{R}}) \vdash^{\prime} M_s : N'' ; r''_s \Longrightarrow \sigma''_t$, then from Lemma 3.12 and Lemma 3.10, we can deduce that $\Delta \triangleright ([\mu_t / s'_i](\sigma''_t)) \equiv_t \sigma_t$. Next, let $R'' = \{s'_i \mapsto r''_s\}$ and $B'' = B' \uplus (\Delta', N'', D', R'')$, then from Rule 18, we have $B'' \vdash^m m'_b : \bullet ; M'_s ; r'''_s \Longrightarrow \bullet$ and $\vdash^{\infty} (M'_s, r'''_s) \Longrightarrow \mu''_t; \sigma'''_t$. But from the premise of Rule 19, if we name $B''' = (\Delta, N\text{of}B, D\text{of}B' \uplus D', R\text{of}B' \uplus R')$, we should also have $B''' \vdash^m m'_b : N' ; \bullet ; r'_s \Longrightarrow \bullet$. Because B'' is related to B''' via a TGC substitution $[\mu_t / s'_i]$ and $\vdash^{\infty} (M'_s, r'_s) \Longrightarrow \mu'_t; \sigma'_t$, we can deduce from Lemma 3.13 that $\Delta \triangleright [\mu_t / s'_i](\mu''_t) \equiv_t \mu'_t :: \kappa'_t$ and $\Delta \triangleright [\mu_t / s'_i](\sigma'''_t) \equiv_t \sigma'_t$. Therefore, we prove $\Delta \triangleright \mu'_t :: \kappa'_t$ and $\Delta; \Gamma \vdash @(f_i[\mu_t])s_i : \sigma'_t$. □

3.4 Examples

To get a better understanding about our algorithm, we show how our translation procedure works on several sample NRC programs. In the following, we try to use the ML syntax whenever possible (instead of the NRC syntax) to simplify the presentation. We omit the stamp part of every NRC identifier by always choosing unique identifier names. We also use ML-like signature declaration as syntactic sugars (keyword `funsig` denotes functor signatures). Our first example is a simple higher-order functor declaration:

```
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,
```

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 12: Translating a simple NRC program

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 12. 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.

Although value components are not included in the NRC calculus, they can be handled easily (see Section 5.1 for more details). In our second example, we show how our algorithm would work on programs that contain value components; we use the example referred earlier in the Introduction section (see Figure 1).

```
signature ASIG = sig type s   val f : s end
funsig FSIG = fsig (X : ASIG) : ASIG
signature BSIG = sig functor F : FSIG end

structure SA : ASIG = struct type s = int   val f = 3 end
structure SB = struct functor F (X : ASIG) : ASIG = struct type s = X.s -> X.s
                                                           fun f (x : X.s) = X.f
                                                    end
               end
functor APP (B : BSIG) : ASIG = B.F(SA)
structure SC = APP(SB)
```

Here, to simplify the presentation, we did not transform the functor body of APP into a definitional structure (m_b); doing so is straightforward because all we need is to enclose the current body inside an additional `struct ... end` layer. The result of the translation is shown in Figure 13. To avoid the notation confusion, a TGC type of form $T(\mu_t)$ is simply represented as μ_t.

signature Z	**type part** (kind κ_Z)	**value part** (type σ_Z)
$X : ASIG$	$\kappa_X = \{s :: \Omega\}$	$\sigma_X = \{f : X_T\}$
$F : FSIG$	$\kappa_F = \kappa_X \to \kappa_X$	$\sigma_F = \forall X_T :: \kappa_X.(\sigma_X \to \{f : B_T.F[X_T]\})$
$B : BSIG$	$\kappa_B = \{F :: \kappa_F\}$	$\sigma_B = \{F : \sigma_F\}$
module Z	**type part** (tycon μ_Z)	**value part** (term e_Z)
SA	$\mu_{SA} = \{s = \texttt{Int}\}$	$e_{SA} = \{f = 3\}$
X	$\mu_X = X_T$	$e_X = X_V$
F	$\mu_F = \lambda X_T :: \kappa_X. \{s = X_T \to X_T\}$	$e_F = \Lambda X_T :: \kappa_X.\lambda X_V : \sigma_X. \{f = \lambda x : (X_T.s).(X_V.f)\}$
SB	$\mu_{SB} = \{F = \mu_F\}$	$e_{SB} = \{F = e_F\}$
B	$\mu_B = B_T$	$e_B = B_V$
APP	$\mu_{APP} = \lambda B_T :: \kappa_B. (B_T.F)[\mu_{SA}]$	$e_{APP} = \Lambda B_T :: \kappa_B.\lambda B_V : \sigma_B. @(B_V.F[\mu_{SA}])e_{SA}$
SC	$\mu_{SC} = \mu_{APP}[\mu_{SB}]$	$e_{SC} = @(e_{APP}[\mu_{SB}])e_{SB}$

Figure 13: Translating an NRC program with value component

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:

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

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

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

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 heavy-weight *control flow analysis* [Shi91] 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 [SA95] and the FLINT/ML compiler [Sha97b]. 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 [MTHM97] 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 this section, we explain how we handle other features in SML'97 and how our techniques interact with separate compilation.

5.1 Value components

Adding value components (e.g., the `val` keyword in ML) to the NRC calculus is quite trivial. Both the semantics and the translation are already equipped with the necessary utility functions to support value specifications and value declarations. 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.

5.2 Functor application

The static semantics given in Section 3.2 always re-elaborates the functor body at each functor application, but much of the work is redundant. For example, given a functor definition F inside the body of another functor G, each time functor G is applied, the body of G thus also the definition of F must be re-elaborated; according to Rule 18 in Figure 9, elaborating the definition of F always require re-elaborating the body of F, but this is unnecessary. Our implementation tries very hard to eliminate all such redundant elaborations.

Value components are also elaborated once and for all at functor definition. More specifically, at each functor application, we do not redo the type-checking of value declarations inside the functor body. Instead, we infer the types of all value components and use the realization environment to propagate type identities. Functor application under our semantics involves re-elaborating the type-related components only.

5.3 Opaque signature matching

The translation algorithm presented in Section 3.3 assumes that signature matching is done transparently. However, opaque signature matching [MTHM97] can be supported quite easily as well by using the same signature-instantiation algorithm presented in Figure 10.

All flexible type components are turned into "abstract" types, represented as fresh stamps annotated with its representation type (e.g., $n(\kappa, \mu)$). Opaque matching creates a structure where all flexible type components are turned into "abstract" types. For example, in the following ML code:

```
structure AA :> sig type s val f : s end = struct type s = real val f = 3.0 end
```

the `s` component in the resulting structure `AA` is a new abstract type. In our stamp-based semantics, `s` will be assigned a new unique type stamp. We also memoize the actual type information for each abstract type, so `s` will be represented as $n(\Omega, \mu)$ where n is the new stamp, Ω is the kind, and $\mu = \mathit{real}$ is the representation

type. This allows the compiler to support type-based manipulations (e.g., representation analysis, pickling, debugging) even on abstract values (e.g., `AA.f`).

The NRC-to-TGC translation converts all stamped types in NRC into concrete TGC types by dropping all the stamps. This is sufficient if the underlying application does not analyze on abstract types themselves. However, more future work is required if we want to propagate abstract types into the intermediate language (e.g., the F_ω calculus must be extended to support abstract types).

5.4 Polymorphic types

Polymorphic functions introduce a small technical problem when propagating the value binding information across the module boundaries. Signature matching may constrain a value of polymorphic type σ into one with a more specific type σ' where σ' is an instance of σ. We solve this problem by always propagating both the value binding information and the original type at its definition site. We can then ignore the change of types during signature matching. Whenever we need to use the binfo, we insert proper wrapper code to coerce it from the original definition type to the new client type.

5.5 Concrete datatypes

Type generativity for datatype declarations can be easily incorporated into our stamp-based semantics. All we need is to generate a new type stamp every time we process a datatype declaration. For the NRC-to-TGC translation, we treat datatype specification as simple type abbreviation; stamps created for datatype declaration are ignored by the translation. This won't be a problem as long as the underlying compiler use same representations for structurally equivalent datatypes.

5.6 Sharing constraints

In Section 3, we have assumed that the SFC-to-NRC translation can convert all sharing constraints into type abbreviations. This is not always possible inside the SML'97 language. If datatype specifications are allowed in signatures. For example, in the following signature,

```
signature XSIG = sig type s
                     type u = s -> s
                     datatype t = A of s * u
                     sharing type s = t
                 end
```

type sharing between `s` and `t` cannot be lifted to the specification site for `s` as it leads to mutually recursive type definitions. Our implementation does not eliminate such type sharing, instead, we use a more sophisticated signature instantiation algorithm to calculate the set of flexible type components in a signature.

5.7 Relationship to separate compilation

Several recent papers [Ler94, HL94] argued that the SML'90 module language [MTH90] provides poor support to Modula-2-like *true separate compilation*, especially under those scenarios described by Cardelli [Car97]. The new SML'97 module language fixed some of these problems by supporting opaque signature matching and type

abbreviations in signatures. Still, an SML'97 program fragment cannot be closed over all of its free functor variables since modules in SML'97 does not allow parameterization over other functors.

Adding higher-order modules to SML'97 would provide better support to separate compilation, but it still does not support true separate compilation. This is because the signature expression used in the NRC calculus is not expressive enough to capture all internal typing properties of a module. As a result, arbitrary modules cannot be abstracted over because they do not always possess a syntactic signature. For example, in the following ML code,

```
signature SIG = sig type t end
functor FAPP(B: sig functor F(X: SIG) : SIG
                    structure A: SIG
                end) = B.F(B.A)
structure R = struct structure R1 = FAPP(struct functor F(X: SIG) = X
                                                structure A : SIG = struct type t=int end
                                         end)
                     structure R2 = FAPP(struct functor F(X: SIG) = struct type t=int end
                                                structure A : SIG = struct type t=bool end
                                         end)
              end,
```

it is impossible to turn R into a functor parameterized by FAPP and still obtain that the components R1.t and R2.t are both equal to `int`.

Nevertheless, fully transparent higher-order modules does make it easier to support Cardelli [Car97]'s scenarios. Consider the following example: a programmer decides to write a large piece of ML software; he first writes down the interfaces for several of his modules, say, `SIG1` and `SIG2` for structure `S1` and `S2`, `FSIG1` and `FSIG2` for functor `F1` and `F2`; then he decides to write the actual implementation code for a module named `S3`. Let's assume that his code for `S3` may refer to all the previous modules such as `S1`, `S2`, `F1` and `F2`, can he compile `S3` without even implementing the other four modules ?

He can decide what he really wants to convey using either transparent and opaque signatures. Suppose `SIG1` is defined as follows:

```
signature SIG1 = sig type t   val f : t -> t end
```

If he marks `SIG1` as opaque, then he is assuming that `t` will be abstract to the rest of the modules. But he could mark `SIG1` as transparent, which could mean that he doesn't care about the actual definition of `t` or simply that `t` could be anything.

In both cases, the compiler can simply implement `S3` as an higher-order functor taking all four modules as its arguments. Later when the implementations of these four modules become available, the linker applies the functor to these modules, producing an implementation for the real `S3`.

One nice thing about our typed cross-module compilation framework is that we do not lose any static binding information exported by `S3`. The full-transparency property guarantees that however the implementation of the other four modules are like, the linker will always be able to recover the same set of value binding information as if `S3` is compiled as a structure knowing the actual implementations of the other four modules.

5.8 Miscellaneous

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 [Sha97a] shows that type-based optimizations can dramatically improve the performance of heavily modularized ML programs.

Another issue is on how to make the implementation of our translation algorithm efficient and scalable. Because the translation makes heavy use of large modtype information, a naive implementation can easily add exponential overhead to the compilation and execution of a program. In a companion paper [SLM98], we present a series of techniques on how to optimize the time and space costs of representing and manipulating large types; we show that our translation algorithm can be implemented with minimum overhead.

6 Related work

Module systems have been an active research area in the past decade. The ML module system was first proposed by MacQueen [Mac86] and later incorporated into Standard ML [MTH90]. Harper and Mitchell [HM93] 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 [HMM90], 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 [HL94, Ler94, Ler95, Lil97, Tof92, MT94, Jon96] and mostly resolved in SML'97 [MTHM97]. The main remaining issue is with the design of higher-order modules, with proposals ranging from fully transparent ones [MT94], to applicative functors [Ler95, Cou97], or abstract functors [HL94, Ler94, Lil97]. Fully transparent modules are most expressive, but it is not clear whether they are absolutely necessary; they also interact poorly with *true separate compilation* [Ler94]. 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 [HMM90, Bis95, Ler95] 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 [HMM90] 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 [Mor97] and also demonstrated in this paper. Harper and Stone [HS97] 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 [Bis95] 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 [Ler95] uses applicative functors to achieve full transparency, but his approach handles limited

[2] Although the paper by Harper *et al* [HMM90] 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 [Mor97] told us that he had problems adapting the original phase-splitting algorithm [HMM90] to SML [MTHM97] 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.

functor arguments only; Courant [Cou97]'s semantics does not have such restriction, but he did not give a translation of his calculus into the F_ω calculus.

Our static semantics for higher-order modules is based on the the same stamp-based approach used in the official definitions for SML'90 [MTH90, MT91] and SML'97 [MTHM97]. The *stamp-based* semantics uses a global name set to model type generativity and type abstraction: a new type is represented as a fresh stamp not used in the current name set. The operational nature of this approach makes it good for efficient implementations but harder for formal reasonings. A more *type-theoretic* approach is to use existential types or manifest types to module type abstraction [Ler94, HL94]. However, none of the existing type-theoretic approaches have been able to model the full-transparency property in the MacQueen-Tofte system.

Both Lillibridge [Lil97] and Leroy [Ler96a] discussed how to add value identities to their module interfaces though neither of them gave any actual algorithm. Blume and Appel [BA97] 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 [SA95] extended Leroy's representation analysis [Ler92] to work for the SML'90 modules; their algorithm works only for the pure-coercion-based representation analysis [Ler92]. The algorithm in this paper translates the module language into the F_ω calculus, so type-based optimizations [Ler92, Sha97a, HM95, MMH96, Tol94] that work for F_ω immediately work for higher-order modules as well.

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 [HM95, Sha97a, Sha97b] 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 [MT94].

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 [Sha97b]. 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

[AH95] Alex Aiken and Nevin Heintze. Constraint-based program analysis. POPL'95 Tutorial, January 1995.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[BA97] Matthias Blume and Andrew 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.

[BHLM94] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian 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.

[Bis95] Sandip 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.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Proc. 24th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 266–277. ACM Press, 1997.

[Cha92] Craig 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.

[Cou97] Judicael Courant. An applicative module calculus. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development: LNCS Vol 1214*, pages 622–636, New York, 1997. Springer-Verlag.

[DM82] Luis Damas and Robin 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.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. ACM SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation*, pages 237–247, New York, June 1993. ACM Press.

[Geo97] Lal George. MLRISC: Customizable and reusable code generators. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, 1997.

[Gir72] J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur.* PhD thesis, University of Paris VII, 1972.

[HL94] Robert Harper and Mark 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.

[HM93] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Trans. on Programming Languages and Systems*, 15(2):211–252, April 1993.

[HM95] Robert Harper and Greg 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.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio 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.

[HS97] Robert Harper and Chris 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.

[Jon91] Neil D. Jones. Partial evaluation. POPL'91, tutorial handout, January 1991.

[Jon96] Mark 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.

[Ler92] Xavier 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.

[Ler94] Xavier 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.

[Ler95] Xavier 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.

[Ler96a] Xavier Leroy. A modular module system. Technical report 2866, INRIA, April 1996.

[Ler96b] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):1–32, September 1996.

[Lil97] Mark 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.

[Mac86] David 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.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proc. 23rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[Mor95] Greg Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. Tech Report CMU-CS-95-226.

[Mor97] Greg Morrisett. Personal Communication, Cornell University, January 1997.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, 1991.

[MT94] David MacQueen and Mads Tofte. A semantics for higher order functors. In *The 5th European Symposium on Programming*, pages 409–423, Berlin, April 1994. Spinger-Verlag.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.

[Nel91] Greg Nelson, editor. *Systems programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rey74] John 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.

[SA95] Zhong Shao and Andrew 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.

[Sha97a] Zhong Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 85–98. ACM Press, June 1997.

[Sha97b] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon Univ., Pittsburgh, Pennsylvania, May 1991. CMU-CS-91-145.

[SLM98] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, page (to appear). ACM Press, September 1998.

[Tof92] Mads 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.

[Tol94] Andrew 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.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, Cambridge, MA, 1993.

A Static semantics for TGC

The typing rules for the F_ω-based TGC calculus are given in Figure 14. The rules for constructor equivalence and type equivalence should also include the the standard rules for equivalence and congruence (omitted in Figure 14 for brevity). The equivalence relation for constructors is consistent with the standard notion of reductions in a simply-typed lambda calculus extended with products, so we can easily see that the constructor calculus is strongly normalizing and every TGC constructor has a unique normal form under the defined equivalence. Finally, the typing rules for terms and declarations are also standard as in the usual polymorphic lambda calculus [Gir72, Rey74, HM93].

B SFC and the SFC-to-NRC translation

The surface module calculus SFC is a subset of SML'97 extended with the MacQueen-Tofte higher-order modules [MT94]. The syntax of SFC is defined in Figure 15. Here, each SFC program consists of a sequence of declarations (d). Symbol t, s, and f denote type, structure, and functor identifiers; p_t, p_s, and p_f are access paths for the type, structure, and functor components in a structure. The SFC calculus includes standard constructs such as signatures (M_s), functor signatures (M_f), structure expressions (m_s), and functor expressions (m_f). Unlike NRC, SFC also supports sharing specification, arbitrary functor body, `let` structure, and more flexible signature subsumption.

To translate from SFC to NRC, we perform a series of syntactic transformations and normalizations on SFC. First, we eliminate the type-sharing constraints by converting them into type abbreviations [Ler96b]. Intuitively, a sharing specification can be pushed through a sequence of other specifications until it hits the root identifier involved in the constraints. For example, a specification such as

$$\texttt{type } t :: \kappa_c \texttt{ sharing type } t = p_t$$

can be rewritten as a type abbreviation:

$$\texttt{type } t :: \kappa_c = p_t\,.$$

Similarly, a specification such as $\texttt{structure } s \texttt{ : sig } D \texttt{ end sharing type } p_t = p'_t$ can push the sharing constraint inside to D if the root identifier of p_t is same as s. Leroy [Ler96b] gave a detailed algorithm for this procedure.

Next, we convert all SFC expressions into *A-normal form* [FSDF93]: functor application must be applying a functor identifier (e.g., f) to a structure identifier (e.g., s) only; signature matching is only performed on structure identifiers as well. For example, functor application $m_f(m_s)$ can be rewritten as

Constructor formation and constructor equivalence: $\boxed{\Delta \triangleright \mu_t :: \kappa_t \text{ and } \Delta \triangleright \mu_t \equiv_t \mu'_t :: \kappa_t}$

$$\frac{}{\Delta \uplus \{\alpha :: \kappa_t\} \triangleright \alpha :: \kappa_t}\ (cvar) \qquad \frac{}{\Delta \triangleright \mathtt{Int} :: \Omega}\ (cint) \qquad \frac{\Delta \triangleright \mu_t :: \Omega \quad \Delta \triangleright \mu'_t :: \Omega}{\Delta \triangleright \mu_t \to \mu'_t :: \Omega}\ (carw)$$

$$\frac{\Delta \uplus \{\alpha :: \kappa_t\} \triangleright \mu_t :: \kappa'_t}{\Delta \triangleright (\lambda\alpha :: \kappa_t.\mu_t) :: \kappa_t \to \kappa'_t}\ (cfn) \qquad \frac{\Delta \triangleright \mu_t :: \kappa'_t \to \kappa_t \quad \Delta \triangleright \mu'_t :: \kappa'_t}{\Delta \triangleright \mu_t[\mu'_t] :: \kappa_t}\ (capp)$$

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

$$\frac{\Delta \uplus \{\alpha :: \kappa'_t\} \triangleright \mu_t :: \kappa_t \quad \Delta \triangleright \mu'_t :: \kappa'_t}{\Delta \triangleright (\lambda\alpha :: \kappa'_t.\mu_t)[\mu'_t] \equiv_t [\mu'_t/\alpha]\mu_t :: \kappa_t}\ (\beta\text{–}eqv) \qquad \frac{\Delta \triangleright \mu_t :: \kappa_t \to \kappa'_t \quad \alpha \notin \mathrm{Dom}(\Delta)}{\Delta \triangleright \lambda\alpha :: \kappa_t.(\mu_t[\alpha]) \equiv_t \mu_t :: \kappa_t \to \kappa'_t}\ (\eta\text{–}eqv)$$

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

Type formation and type equivalence: $\boxed{\Delta \triangleright \sigma_t \text{ and } \Delta \triangleright \sigma_t \equiv_t \sigma'_t}$

$$\frac{\Delta \triangleright \mu_t :: \Omega}{\Delta \triangleright T(\mu_t)}\ (tmono) \qquad \frac{\Delta \triangleright \sigma_t \quad \Delta \triangleright \sigma'_t}{\Delta \triangleright \sigma_t \to \sigma'_t}\ (tarw) \qquad \frac{\Delta \uplus \{\alpha :: \kappa_t\} \triangleright \sigma_t}{\Delta \triangleright \forall\alpha :: \kappa_t.\sigma_t}\ (tpoly)$$

$$\frac{\Delta \triangleright \sigma_t \quad \ldots \quad \Delta \triangleright \sigma'_t}{\Delta \triangleright \{l : \sigma_t, \ldots, l' : \sigma'_t\}}\ (tprod) \qquad \frac{\Delta \triangleright \mu_t :: \Omega \quad \Delta \triangleright \mu'_t :: \Omega}{\Delta \triangleright T(\mu_t \to \mu'_t) \equiv_t T(\mu_t) \to T(\mu'_t)}\ (tarw\text{–}eqv)$$

Term formation and declaration formation: $\boxed{\Delta;\Gamma \vdash e_t : \sigma_t \text{ and } \Delta;\Gamma \vdash d_t : \Gamma'}$

$$\frac{}{\Delta;\Gamma \vdash i : T(\mathtt{Int})}\ (int) \qquad \frac{}{\Delta;\Gamma \vdash x : \Gamma(x)}\ (var) \qquad \frac{\Delta;\Gamma \vdash d_t : \Gamma' \quad \Delta;\Gamma \uplus \Gamma' \vdash e_t : \sigma_t}{\Delta;\Gamma \vdash \mathtt{let}\ d_t\ \mathtt{in}\ e_t : \sigma_t}\ (let)$$

$$\frac{\Delta;\Gamma \uplus \{x : \sigma_t\} \vdash e_t : \sigma'_t}{\Delta;\Gamma \vdash \lambda x : \sigma_t.e_t : \sigma_t \to \sigma'_t}\ (fn) \qquad \frac{\Delta;\Gamma \vdash e_t : \sigma'_t \to \sigma_t \quad \Delta;\Gamma \vdash e'_t : \sigma'_t}{\Delta;\Gamma \vdash @e_t e'_t : \sigma_t}\ (app)$$

$$\frac{\Delta \uplus \{\alpha :: \kappa_t\};\Gamma \vdash e_t : \sigma_t}{\Delta;\Gamma \vdash \Lambda\alpha :: \kappa_t.e_t : \forall\alpha :: \kappa_t.\sigma_t}\ (tfn) \qquad \frac{\Delta;\Gamma \vdash e_t : \forall\alpha :: \kappa_t.\sigma_t \quad \Delta \triangleright \mu_t :: \kappa_t}{\Delta;\Gamma \vdash e_t[\mu_t] : [\mu'_t/\alpha]\sigma_t}\ (tapp)$$

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

$$\frac{}{\Delta;\Gamma \vdash \varepsilon : \emptyset_\Gamma}\ (edec) \qquad \frac{\Delta;\Gamma \vdash e_t : \sigma_t \quad \Delta;\Gamma \uplus \{x : \sigma_t\} \vdash d_t : \Gamma'}{\Delta;\Gamma \vdash (x = e_t); d_t : \{x : \sigma_t\} \uplus \Gamma'}\ (sdec)$$

Figure 14: Static semantics for TGC

Access paths:

strpath	p_s	::=	$s \mid p_s.s$
fctpath	p_f	::=	$f \mid p_s.f$
typpath	p_t	::=	$t \mid p_s.t$

Signatures and specifications:

spec	D	::=	$\varepsilon \mid DD'$
		$\mid$	`type` $t :: \kappa_c$
		$\mid$	`type` $t :: \kappa_c = \mu_c$
		$\mid$	`structure` $s : M_s$
		$\mid$	`functor` $f : M_f$
		$\mid$	D `sharing type` $p_t = p'_t$
sig	M_s	::=	`sig` D `end`
fsig	M_f	::=	`fsig`$(s:M_s)M'_s$

Kinds and types:

kind	κ_c	::=	$\Omega \mid \Omega \rightarrow \kappa_c$
tycon	μ_c	::=	$p_t \mid int \mid \mu_c \rightarrow \mu_c{}'$
		$\mid$	$\lambda t::\Omega.\mu_c \mid \mu_c[\mu_c{}']$

Modules and declarations:

decl	d	::=	$\varepsilon \mid dd' \mid$ `local` d `in` d' `end`
		$\mid$	`type` $t :: \kappa_c = \mu_c$
		$\mid$	`structure` $s = m_s$
		$\mid$	`functor` $f = m_f$
str	m_s	::=	$p_s \mid m_f(m_s) \mid (m_s : M_s)$
		$\mid$	`struct` d `end`
		$\mid$	`let` d `in` m_s `end`
fct	m_f	::=	$p_f \mid$ `funct` $(s:M_s)m_s$

Figure 15: Syntax of the surface module calculus SFC

```
let functor f = m_f
    structure s = m_s
 in f(s)
end.
```

This procedure is recursively applied to all SFC sub-expressions.

Third, in order to avoid name captures, we attach a unique internal name to each SFC identifier; we also make sure that all functor definitions and functor signatures have uniquely named formal parameters. More specifically, each time we bind an identifier x, we generate a new stamp i and rename all the uses of this x into x_i. For example, the following program (assume it is already A-normalized):

```
structure s =
  let structure u = struct type t = int   type x = t end
   in (u : sig type t end)
  end
```

is rewritten into:

```
structure s_i =
  let structure u_j = struct type t_k = int   type x_m = t_k end
   in (u_j : sig type t_l end)
  end
```

where the t component of structure s is now referenced as $s_i.t_l$ (but not $s_i.t_k$). This renaming scheme creates a name mismatch for some signature-matching expressions (e.g., between the t_k of u_j and t_l in the signature). We fix this by introducing an auxiliary structure definition u_n:

```
structure s_i =
  let structure u_j = struct type t_k = int   type x_m = t_k end
      structure u_n = struct type t_l = u_j.t_k end
   in (u_n : sig type t_l end)
  end
```

The renaming algorithm can be implemented by maintaining an environment that maps every type identifier into one with proper internal stamp, every structure (or functor) identifier into a signature (or functor signature) whose components have already been properly stamped. Every access path p_x can then be renamed by a simple environment lookup.

Along with the renaming procedure, we also make concrete the *enrichment* coercions occurred at every signature matching—including those performed on the argument and result of each functor application. This is reasonable because we often have to create a new structure such as u_n anyway to eliminate name mismatches; for example, the x component in the above structure u_j is dropped in u_n. This procedure is recursively performed on all components; for instance, matching a functor `functor` f = `funct`$(s : M_s)m_s$ against a functor signature `fsig`$(s' : M'_s)M''_s$ will result in the following coercion:

```
functor f'(s' : M'_s) =
  let structure a = (s' : M_s)
      structure r = f(a)
      structure c = (r : M''_s)
   in c
  end
```

After this transformation, signature matching such as $(s_i : M_s)$ always satisfies the following invariants: structure s_i contains the same number of components as those in M_s; each pair of corresponding components have same identifiers and stamps (so they are ordered in the same way); furthermore, each functor component in s_i must have a signature syntactically equivalent to its counterpart in M_s.

Fourth, we convert the functor body into a definitional structure expression of form `struct`...`end`. Given a functor expression `funct` $(s_i : M_s)m_s$, we wrap an extra layer around the functor body m_s into:

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

Functor body in functor signatures is 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, we lift out all module declarations in the `let` expressions and turn them into `local` declarations. This is always possible if we assume an SFC program is just a set of module declarations. Each declaration inside the module expressions can be lifted upwards until one of the following happens: (1) it reaches the top of a functor body, but after previous transformations, all functor body is of form `struct` d `end`; or (2) it reaches the top-level declaration d of a program. In both cases, we can convert `let` declarations (say d') inside d into `local` declarations by simply attaching them to the front of d as in `local` d' `in` d `end`.