A Flattening Strategy for SML Module Compilation and Its Implementation

Liu Bochao  Atsushi Ohori

This paper presents a method to compile the Standard ML module language into a flattened intermediate language. An innovative point on this approach lies in viewing a functor as a code template with place holders representing the functor arguments. Each functor application fills these place holders in the code template with the actual functor argument and generates a fresh structure. After this compilation, module language constructs are all eliminated. This method allows us to compile the full set of Standard ML language into a typed intermediate language that contains no special mechanism for module, and provides a simpler model for separate compilation. The proposed compilation method has been successfully implemented in the SML♯ compiler, which demonstrates the feasibility of the method. This paper also reports the details of our implementation.

1 Introduction

Standard ML supports large scale programming through its powerful module system. A large Standard ML program typically consists of many modules. Each module can contain not only dynamic resources such as functions but also static resources such as type and exception declarations. This makes the Standard ML module system a powerful tool for organizing a large program in a hierarchical way. However, this feature makes the module system itself a complicated language with an elaborate type system. In fact, the definition of the module language specified in [6] is more involved than that of the core language. As being such an elaborate language, compilation of a module language into efficient code is a nontrivial task, and several approaches have been proposed.

MacQueen et al. [7] proposed a compilation method where a structure is compiled into a record consisting of its dynamic components, and a functor is compiled into a function closure mapping record to record. Since the module language is compiled into the combination of core language constructs, this strategy yields a simple compilation method that does not require any additional machinery but those already required by the core language. A major weakness to this strategy is that each module construct yields a term that needs to be evaluated at runtime and yields a heap structure. Although this might be inevitable for languages such as those proposed in [16][17] where modules are first class objects that can be manipulated at runtime, for the Standard ML module language where modules are static constructs, this dynamic overhead is certainly undesirable. This would be particularly problematic for a large system consisting of a large collection of modules.

A module flattening compilation strategy proposed by Elsman [8][9] solves the problem of heap allocation for modules, where a structure is flattened, a functor definition is elaborated only for type-checking and code generation is postponed to
functor application time. More specifically, a functor declaration in the module language behaves like a macro declaration, which is type checked but not flattened and kept in the compiler’s static environment in the form of the input source code for type checking. That is, the functor body waits for type checking and flattening again in the future. Thus each time at functor application, the compiler first type checks and flattens the functor argument and generates a binding for the argument, and then fetches the functor body from the static environment and type checks and flattens the stored body under the bindings for functor actual argument. Note that if the functor is applied several times, then each time the functor body needs to be type checked and flattened again and again. This strategy may be desired for MLKit [2] compiler to get the precise region related property for functor body, and for MLton compiler [3] to do whole program analysis. But drawbacks to this approach is that it incurs nontrivial compilation time overhead and does not support true separate compilation. In true separate compilation, the functor should be treated like a separately compiled unit, and its body should be compiled as an object code so that at functor application the compiler only links the unresolved external references with the real ones.

In practical system development, separate compilation plays an important role in efficient and systematic system development. The ability to link separately compiled modules is also a key to achieve interoperability with other languages. Thus SML♯ [1] aims at providing such a facility.

To design a module language compilation that can scale up to true separate compilation, we refine the flattening strategy of Elsman [8] by rewriting the functor flattening algorithm so that it compiles the functor body when it is defined. The idea is to compile a functor body as a “code template” containing holes representing the functor formal argument. At functor application, the compiler performs template instantiation by filling the holes with the actual argument structures. This process is similar to the program linking that merges separate object files into an executable unit in which external references between linkage units are resolved. Based on this strategy, we have developed a flattening compilation algorithm for SML♯, a conserved extension of the Standard ML, and have fully implemented a module compiler. This method goes closer to the separate compilation model. We believe this approach helps build our separate compilation framework in the near future.

Although the idea underlying the flattening compilation is simple and intuitive, there are several subtle issues. One is to maintain proper name binding. Signature curtailment, functor generativity are all involved in the name binding compilation. In addition, since SML♯ supports the interactive evaluation mode, besides the unique variable allocation, global variables across compilation sessions entail the adoption of global arrays to hold values for such variables. Thus the compiler needs to maintain two representations of variables, the unique variables and the global array indices. The other issue is how to deal with type manipulation entailed by SML♯’s overall type directed compilation strategy [13] that enables some powerful extensions of SML♯ [14][15] and its interoperability feature [19]. Signature possibly narrower type restrictions, abstract types in the opaque signature and the functor argument are all involved in the type passing code generation.

We have worked out all the issues and have developed a module compiler for the SML♯ compiler. The compiler has been tested with the Basis Library of Standard ML and various benchmarks such as barnes, boyer, coresml, count, graphs, fft, knuth, bendix, lexgen, life, logic, mandelbrot, myacc, nucleic, ratio, regions, ray, simple, tsp and vlw, which demonstrates the feasibility of our proposed method.

The following sections are organized as follows. Section 2 presents the source language and its type system. Section 3 gives our module compilation algorithm. Section 4 establishes the type soundness property of our compilation strategy. Section 5 presents the implementation in our SML♯ compiler. Section 6 shows the benchmark. Section 7 concludes the article.

2 Source Language

To present our method, we define a simplified source language given in Figure 1, which is a subset of the Standard ML language [6].
Core Language Syntax

\[
\begin{align*}
\text{ty} & ::= \text{lt} \mid \alpha \mid \text{ty}_1 \rightarrow \text{ty}_2 \\
\text{exp} & ::= \text{lv} \mid \text{exp} \text{ exp} | \text{fn} \text{ lv} \Rightarrow \text{exp} \mid \text{let} \text{ dec in exp} \\
\text{dec} & ::= \text{val} \ v = \text{exp} \mid \text{type} \ t = \text{ty} \mid \text{open} \ \text{ls}
\end{align*}
\]

Module Language Syntax

\[
\begin{align*}
sigexp & ::= \text{sig spec end} \\
\text{spec} & ::= \text{val} \ v : \sigma \mid \text{type} \ t \mid \text{structure} \ s : \sigexp \mid \text{spec spec} \mid \sigexp \text{ where type} \ \text{lt} = \text{ty} \\
\text{strexp} & ::= \text{struct} \ \text{strored} \ \text{end} \mid \text{ls} \mid f(\text{strexp}) \mid \text{strexp} : \sigexp \\
\text{strored} & ::= \text{dec} \mid \text{structure} \ s = \strexp \mid \text{strexp} \ \text{strored}
\end{align*}
\]

\[
\begin{align*}
\text{topdec} & ::= \text{strored} \mid \text{functor} \ f(s : \sigexp) = \strexp \mid \text{topdec} \ \text{topdec}
\end{align*}
\]

Fig. 1 The Source Language Syntax

2.1 The Syntax

We let \(TId, VId, StrId\), and \(FunId\) be given countably infinite sets of type identifiers (ranged over by \(t\)), value identifiers (ranged over by \(v\)), structure identifiers (ranged over by \(s\)), functor identifiers (ranged over by \(f\)), respectively. Type identifiers, value identifiers and structure identifiers are qualified with a sequence of structure identifiers separated with a period “.” to form so-called long identifiers. We let \(ls, lt\) and \(lv\) range over long structure identifiers\((LongStrId)\), long type identifiers\((LongTId)\) and long value identifiers\((LongVId)\), whose syntax are given below.

\[
\begin{align*}
ls & ::= s \mid s.\text{ls} \\
lt & ::= t \mid s.\text{lt} \\
\text{lv} & ::= v \mid s.\text{lv}
\end{align*}
\]

2.2 Type System

We roughly follow [6] and give the type system of our language as an interpretation of the source language into semantic objects.

Figure 2 gives the semantic objects, long identifier access functions and the functionality of the static interpretation relations.

When \(S'\) and \(S''\) are sets, \(S' \xrightarrow{\text{fin}} S''\) denotes the set of finite maps(partial functions with finite domain) from \(S'\) to \(S''\).

We use \(\alpha\) to range over a countably infinite set of type variables\((TyVar)\), \(tn\) to range over a countably infinite set of type names\((TyName)\), \(\tau\) to range over types\((Type)\) and \(\sigma\) to range over type schemes\((TypeScheme)\). \(TyNameSet\) is used to denote the set of all type name sets. For any object \(A\), \(\text{tyvars}(A)\) is the set of type variables free in \(A\), and \(\text{tynames}(A)\) is the set of free type names in \(A\).

Type variables are bound in type schemes by \(\forall\) in the ordinary sense. A type scheme \(\sigma\) is more general than \(\sigma'\), written \(\sigma \succ \sigma'\), if \(\sigma = \forall(\alpha_1, \ldots, \alpha_n).\tau\), \(\sigma' = \forall(\beta_1, \ldots, \beta_m).\tau'\), \(\tau'\) is obtained from \(\tau\) by substituting each \(\alpha_i\) with some type and \(\{\beta_1, \ldots, \beta_m\}\) contains no free type variable of \(\sigma\). An environment \(E\) or \((TE, VE, SE)\) consists of a triple of a type environment \(TE\), a value environment \(VE\) and a structure environment \(SE\). A basis \(B\) or \((F, E)\) consists of a pair of a functor environment \(F\) and an environment \(E\). A signature \(\Sigma\) or \((T)E\) consists of a bound type name set \(T\) and an environment \(E\). A functor signature \(\Phi\) or \((T)(E_1 \rightarrow E_2)\) consists of a bound type name set \(T\), a functor argument environment \(E_1\) and a functor body environment \(E_2\).

A realization is a map \(\varphi : TyName \rightarrow Type\). The support \(\text{Supp}\ \varphi\) is the set of type name \(tn\) such that \(\varphi(tn) \neq tn\). The realization is extended to any object with the effect that it replaces each type name \(tn\) by \(\varphi(tn)\). An environment \(E'\) is an instance of a signature \((T)E\), written \((T)E \geq E'\), if there exists \(\varphi\) such that \(\varphi(E) = E'\) and \(\text{Supp}\ \varphi \subseteq T\). Given \(\Phi = (T_1)(E_1, \Sigma_1)\), \((E_2, \Sigma_2)\) is an instance of \(\Phi\), written \(\Phi \geq (E_2, \Sigma_2)\), if there exists \(\varphi\) such that \(\varphi(E_1, \Sigma_1) = (E_2, \Sigma_2)\) and \(\text{Supp}\ \varphi \subseteq T_1\). A prefix \(T\) in \((T)A\) for any object \(A\) is the bound type name set of that object.

Let \(E_1 = (TE_1, VE_1, SE_1)\) and \(E_2 = (TE_2, VE_2, SE_2)\). An environment \(E_1\) enriches another \(E_2\), written \(E_1 \succ E_2\), if

\[
\begin{align*}
&\bullet \ \text{Dom}(TE_1) \supseteq \text{Dom}(TE_2), \text{ and } TE_1(t) = TE_2(t) \text{ for all } t \in \text{Dom}(TE_2). \\
&\bullet \ \text{Dom}(VE_1) \supseteq \text{Dom}(VE_2), \text{ and } VE_1(v) \succ VE_2(v) \text{ for all } v \in \text{Dom}(VE_2).
\end{align*}
\]
Types and Typing Environments

- **type:**
  \[ \tau ::= \alpha \mid \tau \mid \tau \rightarrow \tau \]

- **type scheme:**
  \[ \sigma ::= \forall \alpha_1 \ldots \alpha_n \mid \tau \]

- **typing basis:**
  \[ B \text{ or } (F, E) \in T\text{Basis} = TFunEnv \times TEnv \]

- **typing environment:**
  \[ E \text{ or } (TE, VE, SE) \in TEnv = TTyEnv \times TValEnv \times TStrEnv \]

- **typing type environment:**
  \[ TE \in TTyEnv = TId \overset{\text{fin}}{\rightarrow} Type \]

- **typing value environment:**
  \[ VE \in TValEnv = VId \overset{\text{fin}}{\rightarrow} TypeScheme \]

- **typing structure environment:**
  \[ SE \in TStrEnv = StrId \overset{\text{fin}}{\rightarrow} TEnv \]

- **typing functor environment:**
  \[ F \in TFunEnv = FunId \overset{\text{fin}}{\rightarrow} TFunSig \]

- **typing signature:**
  \[ (T)E \text{ or } \Sigma \in TSig = TyNameSet \times TEnv \]

- **typing functor signature:**
  \[ \Phi \text{ or } (T_1)(E_1 \rightarrow E_2) \in TFunSig = TyNameSet \times (TEnv \times TEnv) \]

Access functions for long identifiers

- \( \text{lookUp}(t, (TE, VE, SE)) = TE(t) \)
- \( \text{lookUp}(v, (TE, VE, SE)) = VE(v) \)
- \( \text{lookUp}(s.lt, (TE, VE, SE)) = \text{lookUp}(lt, SE(s)) \)
- \( \text{lookUp}(s.lv, (TE, VE, SE)) = \text{lookUp}(lv, SE(s)) \)
- \( \text{lookUp}(s, (TE, VE, SE)) = SE(s) \)
- \( \text{lookUp}(s.ls, (TE, VE, SE)) = \text{lookUp}(ls, SE(s)) \)

Typing Relations

<table>
<thead>
<tr>
<th>environment ⊢ syntax</th>
<th>static semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \vdash ty )</td>
<td>( \Rightarrow \tau )</td>
</tr>
<tr>
<td>( E \vdash exp )</td>
<td>( \Rightarrow \tau )</td>
</tr>
<tr>
<td>( E \vdash dec )</td>
<td>( \Rightarrow E )</td>
</tr>
<tr>
<td>( B \vdash sigexp )</td>
<td>( \Rightarrow \Sigma )</td>
</tr>
<tr>
<td>( B \vdash spec )</td>
<td>( \Rightarrow \Sigma )</td>
</tr>
<tr>
<td>( B \vdash strexp )</td>
<td>( \Rightarrow E )</td>
</tr>
<tr>
<td>( B \vdash strdec )</td>
<td>( \Rightarrow E )</td>
</tr>
<tr>
<td>( B \vdash topdec )</td>
<td>( \Rightarrow B )</td>
</tr>
</tbody>
</table>

Fig. 2 Types, Environments, Access Functions and Typing Relations

- \( \text{Dom}(SE_1) \supseteq \text{Dom}(SE_2) \), and \( SE_1(s) \succ_{SE_2(s)} \) for all \( s \in \text{Dom}(SE_2) \).

An environment \( E \) matches a signature \( \Sigma \) iff there exists \( E' \) such that \( \Sigma \geq E' \prec E \). Note that for simplicity we only consider type declarations excluding datatypes, thus only type equivalence is required between types, i.e. \( TE_1(t) = TE_2(t) \).

The domain and range of a finite map, \( m \), are denoted \( \text{Dom}(m) \) and \( \text{Ran}(m) \). We write \( E_1 + E_2 \) and \( B_1 + B_2 \) to mean \( E_1 \) and \( B_1 \) are overridden by \( E_2 \) and \( B_2 \) respectively.

Using these definitions, the type system of the
source language is given by the set of static interpretation relations. Figure 3 and 4 give the interpretation rules for the core language and the module language.

3 Flattening Module Compilation

A compilation algorithm is given as a refinement to the standard static interpretation given in the previous section. The major role of module compilation is to map long identifiers in the source language to variables in the target language. This is done by refining the mapped object from long identifiers to include variables in the target language.

Figure 5 gives the set of semantic objects of our module compilation and the functionality of the compilation relations. We omit the similar long identifier access functions as before.

Let Var (ranged over by \(x\)) be a given countably infinite set of variables and VarSet be the set of all variable sets. Let NameSet (ranged over by \(N\)) be the set of all such set whose element is either a variable \(x\) or a type name \(tn\). For any object \(A\), names of \(A\) is the set of variables and type names free in \(A\). \((\sigma_1, x : \sigma_2)\) is a typed variable where \(\sigma_1\) represents the type scheme used in the type checking of the source language and \(\sigma_2\) is the actual implementation type used in the type checking of the target language. Note that \(\sigma_2\) is different from \(\sigma_1\) at the signature constraint spot. For example,

\[
\text{struct }\quad \text{val f = fn x => x end}
\]

After compilation for this signature constrained structure, the typed variable of \(f\) is \((\text{int} \rightarrow \text{int}, (f : \forall \alpha. \alpha \rightarrow \alpha))\).

\(\forall \mathcal{E}\) is a compilation value environment associating each value identifier with a typed variable. \(\mathcal{SE}\) is a compilation structure environment associating each structure identifier with a compilation environment that in turn consists of a compilation value environment and a compilation structure environment. \((N_1)(\mathcal{E}_1 \rightarrow (N_2)(\mathcal{E}_2, d))\) is a code template with \(N_1\) capturing both the type names and variables in the functor formal argument and \(N_2\) capturing the generated variables in the functor body, \(\mathcal{E}_1\) representing the functor argument compilation environment, \(\mathcal{E}_2\) denoting the functor body compilation environment and \(d\) representing the intermediate code of functor body. For any object \(A\), the prefix \(N\) binds names in \((N).A\). \(\mathcal{F}\) is a functor compilation environment associating each functor identifier with a code template. \(\mathcal{B}\) is a compilation basis consisting of a functor compilation environment and a compilation environment.

Now realization is a map \(\varphi^+: (TyName \rightarrow Type) \cup (Var \rightarrow Var)\). The realization is extended to any object with the effect that it replaces each type name \(tn\) by \(\varphi^+(tn)\), each variable \(x\) by \(\varphi^+(x)\). The support \(\text{Supp } \varphi^+\) is the set of name \(tn\) or \(x\)
For sigexp:

\[ B \vdash \text{spec} \Rightarrow \Sigma \]

\[ E \vdash ty \Rightarrow \tau \quad (F, E) \vdash \text{sigexp} \Rightarrow (T) E \]

\[ \text{lookUp}(lt, E_1) = tn \quad \text{tn} \in T \quad \varphi = \{ \text{tn} \mapsto \tau \} \]

\[ F, E \vdash \text{sigexp} \text{ where type } \text{lt} = ty \Rightarrow (T)(\varphi(E)) \]

For spec:

\[ B \vdash \text{type } t \Rightarrow \{(t \mapsto \text{tn}), \{\}, \{\} \} \text{ (tn fresh)}\]

\[ E \vdash \text{ty} \Rightarrow \tau \quad \text{tyvars}(\tau) = \{\alpha_1, \ldots, \alpha_n\} \]

\[ B \vdash \text{val } v: \text{ty} \Rightarrow \{(\{}\{\}, \{v \mapsto \forall \alpha_1 \ldots \alpha_n. \tau\}, \{\}\} \]

\[ B \vdash \text{sigexp} \Rightarrow (T)E \]

\[ B \vdash \text{structure } s:\text{sigexp} \Rightarrow (T)(\{\}, \{\}, \{s \mapsto E\}) \]

\[ B \vdash \text{spec}\_1 \Rightarrow (T\_1)E_1 \quad B + (\{\}, E_1) \vdash \text{spec}\_2 \Rightarrow (T\_2)E_2 \quad \text{Dom}(E_1) \cap \text{Dom}(E_2) = \emptyset \]

\[ B \vdash \text{spec}\_1 \text{ spec}\_2 \Rightarrow (T\_1 \cup T\_2)(E_1 + E_2) \]

For strexp:

\[ B \vdash \text{strdec} \Rightarrow E_1 \]

\[ B \vdash \text{struct strdec end} \Rightarrow E_1 \]

\[ (F, E) \vdash \text{strexp} \Rightarrow E_1 \quad F(f) \succ E_2 \Rightarrow E_3' \quad E_1 \succ E_2' \]

\[ (F, E) \vdash f(\text{strexp}) \Rightarrow E_3' \]

\[ B \vdash \text{strexp} \Rightarrow E_1 \quad B \vdash \text{sigexp} \Rightarrow \Sigma \quad \Sigma \geq E_2 \times E_1 \]

\[ B \vdash \text{strexp: sigexp} \Rightarrow E_2 \]

For strdec:

\[ E \vdash \text{dec} \Rightarrow E \]

\[ (F, E) \vdash \text{dec} \Rightarrow E \]

\[ B \vdash \text{strexp} \Rightarrow E_1 \]

\[ B \vdash \text{structure } s = \text{strexp} \Rightarrow (\{\}, \{\}, \{s \mapsto E_1\}) \]

\[ (F, E) \vdash \text{strdec}\_1 \Rightarrow E_1 \quad (F, E + E_1) \vdash \text{strdec}\_2 \Rightarrow E_2 \]

\[ (F, E) \vdash \text{strdec\_1 strdec\_2} \Rightarrow E_1 + E_2 \]

For topdec:

\[ B \vdash \text{strdec} \Rightarrow E_1 \]

\[ B \vdash \text{strdec} \Rightarrow (\{\}, E_1) \]

\[ (F, E) \vdash \text{sigexp} \Rightarrow (T)E_1 \quad (F, E + (\{\}, \{\}, \{s \mapsto E_1\})) \vdash \text{strexp} \Rightarrow E_2 \]

\[ (F, E) \vdash \text{functor } f(s:\text{sigexp}) \Rightarrow [(f \mapsto (T)(E_1 \rightarrow E_2)), (\{\}, \{\}, \{\})] \]

\[ B \vdash \text{topdec}\_1 \Rightarrow B_1 \quad B + B_1 \vdash \text{topdec}\_2 \Rightarrow B_2 \]

\[ (F, E) \vdash \text{functor } f(s:\text{sigexp}) \Rightarrow [(f \mapsto (T)(E_1 \rightarrow E_2)), (\{\}, \{\}, \{\})] \]

\[ B \vdash \text{topdec\_1 topdec\_2} \Rightarrow B_1 + B_2 \]

\[ \text{Fig. 4 \ The Typing Rules for Modules} \]

such that \( \varphi^+(\text{tn}) \neq \text{tn} \) and \( \varphi^+(x) \neq x \).

Given a functor template \( C = (N_1)(\mathcal{E}_1, (N_2)(\mathcal{E}_2, d)) \), \( (\mathcal{E}_3, (N_3)(\mathcal{E}_4, d_1)) \) is an instance of \( C \), written \( C \geq (\mathcal{E}_3, (N_3)(\mathcal{E}_4, d_1)) \), if there exists \( \varphi^+ \) such that \( \varphi^+(\mathcal{E}_1, (N_2)(\mathcal{E}_2, d)) = (\mathcal{E}_3, (N_3)(\mathcal{E}_4, d_1)) \) and \( \text{Supp} \varphi^+ \subseteq N_1 \).

The generalization relation \( \sigma \succ \sigma' \) on types naturally extends to typed variables as follows. A typed variable \( (\sigma_1, x_1 : \sigma_1') \) generalizes \( (\sigma_2, x_2 : \sigma_2') \), written \( (\sigma_1, x_1 : \sigma_1') \succ (\sigma_2, x_2 : \sigma_2') \), if \( \sigma_1 \succ \sigma_2 \) and \( (x_1 : \sigma_1') = (x_2 : \sigma_2') \). Enrichment relation on compilation environments, \( \mathcal{E}_1 \succ \mathcal{E}_2 \), is defined using this relation, which is similar as the enrichment relation for typing environment before.

Given two environments \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \), the type instantiation function \( \text{inst}(\mathcal{E}_1, \mathcal{E}_2) \) results in a pair \( (N, d) \) of a name set and a sequence of intermediate declarations each of which is in the form of \( \text{val } x_2 : \alpha_1 \ldots \alpha_n. \tau = x_1 \), where \( x_1 \)'s and \( x_2 \)'s are all the corresponding variables in \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \) such that \( \text{lookUp}(lv, \mathcal{E}_1) = (\sigma_1, x_1 : \sigma_1') \) and \( \text{lookUp}(lv, \mathcal{E}_2) = (\forall \alpha_1 \ldots \alpha_n. \tau, x_2 : \sigma_2') \) for any \( lv \),
Module Compilation Target Codes and Environments

target expressions:
\[ e ::= x \mid \lambda x : \tau . e \mid \text{let } d \text{ in } e \]
target declarations:
\[ d ::= e \mid \text{val } x : \alpha_1 \ldots \alpha_n . \tau = e \quad d \]
basis:
\[ \mathcal{B} \text{ or } (\mathcal{F}, \mathcal{E}) \in \mathcal{CBasis} = \mathcal{CFunEnv} \times \mathcal{CEnv} \]
compilation functor environments:
\[ \mathcal{F} \in \mathcal{CFunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{CodeTemplate} \]
compilation environments:
\[ \mathcal{E} \in \mathcal{CEnv} = \mathcal{TTyEnv} \times \mathcal{CValEnv} \times \mathcal{CStrEnv} \]
compilation value environments:
\[ \forall \mathcal{E} \in \mathcal{CValEnv} = \text{VId} \xrightarrow{\text{fin}} \text{TypedVars} \]
compilation structure environments:
\[ \mathcal{SE} \in \mathcal{CStrEnv} = \text{StrId} \xrightarrow{\text{fin}} \mathcal{CEnv} \]
compilation signatures
\[ N(\mathcal{E}) \in \mathcal{CSig} = \text{NameSet} \times \mathcal{CEnv} \]
functor code templates:
\[ C \text{ or } (N_2)((\mathcal{E}_1 \rightarrow (N_2)(\mathcal{E}_2, d)) \in \text{CodeTemplate} = \text{NameSet} \times \mathcal{CEnv} \times \text{NameSet} \times \mathcal{CEnv} \times \text{Dec} \]
typed variables:
\[ (\sigma, x : \sigma') \in \text{TypedVars} = \text{TypeScheme} \times \text{VarSet} \times \text{TypeScheme} \]

Compilation Relations

<table>
<thead>
<tr>
<th>environment ⊢ syntax ⇒ semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \mathcal{E} \vdash \mathcal{ty} \Rightarrow \tau ]</td>
</tr>
<tr>
<td>[ \mathcal{E} \vdash \mathcal{exp} \Rightarrow (\tau, e) ]</td>
</tr>
<tr>
<td>[ \mathcal{E} \vdash \mathcal{dec} \Rightarrow (N, \mathcal{E}, d) ]</td>
</tr>
<tr>
<td>[ \mathcal{B} \vdash \mathcal{sigexp} \Rightarrow (N)\mathcal{E} ]</td>
</tr>
<tr>
<td>[ \mathcal{B} \vdash \mathcal{spec} \Rightarrow (N)\mathcal{E} ]</td>
</tr>
<tr>
<td>[ \mathcal{B} \vdash \mathcal{strexp} \Rightarrow (N, \mathcal{E}, d) ]</td>
</tr>
<tr>
<td>[ \mathcal{B} \vdash \mathcal{strdec} \Rightarrow (N, \mathcal{E}, d) ]</td>
</tr>
<tr>
<td>[ \mathcal{B} \vdash \mathcal{topdec} \Rightarrow (B, d) ]</td>
</tr>
</tbody>
</table>

Fig. 5 Module Compilation Target Codes and Environments, Compilation Relations

For \( \text{ty} \):
\[ \text{lookUp}(\text{lt}, \mathcal{E}) = \tau \]
\[ \mathcal{E} \vdash \mathcal{ty} \Rightarrow \tau \]
\[ \mathcal{E} \vdash \mathcal{ty} \rightarrow \mathcal{ty}_1 \Rightarrow \tau_1 \quad \mathcal{E} \vdash \mathcal{ty}_2 \Rightarrow \tau_2 \]
\[ \mathcal{E} \vdash \mathcal{ty}_1 \rightarrow \mathcal{ty}_2 \Rightarrow \tau_1 \rightarrow \tau_2 \]

For \( \text{exp} \):
\[ \text{lookUp}(\text{lv}, \mathcal{E}) = (\sigma, x : \sigma') \quad \text{σ} \Rightarrow \tau \]
\[ \mathcal{E} \vdash \text{lv} \Rightarrow (\tau, x) \]
\[ \mathcal{E} \vdash \text{exp}_1 \Rightarrow (\tau_1 \rightarrow \tau_2, e_1) \quad \mathcal{E} \vdash \text{exp}_2 \Rightarrow (\tau_1, e_2) \]
\[ \mathcal{E} \vdash \text{exp}_1 \cdot \text{exp}_2 \Rightarrow (\tau_2, e_1, e_2) \]
\[ \mathcal{E} \vdash \text{dec} \Rightarrow (N, \mathcal{E}_1, d) \quad \mathcal{E} \vdash \text{exp} \Rightarrow (\tau, e) \]
\[ \mathcal{E} \vdash \text{let } \text{dec } \text{in } \text{exp} \Rightarrow (\tau, \text{let } d \text{ in } e) \]

For \( \text{dec} \):
\[ \mathcal{E} \vdash \text{exp} \Rightarrow (\tau, e) \quad \{\alpha_1, \ldots, \alpha_n\} \cap \text{tyvars}(\mathcal{E}) = \emptyset \quad \sigma = \forall (\alpha_1, \ldots, \alpha_n).\tau \quad (x \text{ fresh}) \]
\[ \mathcal{E} \vdash \text{val } v = \text{exp} \Rightarrow (\{x\}, (\{\}, \{v \mapsto (\tau, x : \tau)\}, (\{\}), \text{val } x : \alpha_1 \ldots \alpha_n . \tau = e) \]
\[ \mathcal{E} \vdash \mathcal{ty} \Rightarrow \tau \]
\[ \mathcal{E} \vdash \text{type } t = \text{ty} \Rightarrow (\{\}, (\{t \mapsto \tau\}, (\{\}, (\{\}), \text{e})) \]
\[ \text{lookUp}(\text{ls}, \mathcal{E}) = \mathcal{E}' \]
\[ \mathcal{E} \vdash \text{open } \text{ls } \Rightarrow (\{\}, \mathcal{E}', e) \]

Fig. 6 Compilation Rules for the Core Language
For `sigexp`:

\[
\begin{align*}
B \vdash \text{spec} &\Rightarrow (N)\mathcal{E} \\
B \vdash \text{sig spec end} &\Rightarrow (N)\mathcal{E}
\end{align*}
\]

\[
\begin{align*}
\mathcal{E} \vdash ty &\Rightarrow \tau \\
(F,\mathcal{E}) \vdash \text{sigexp} &\Rightarrow (N)\mathcal{E}_1 \\
\text{lookUp}(lt,\mathcal{E}_1) &\Rightarrow tn \quad tn \in N \\
\varphi &\Rightarrow \{tn \mapsto \tau\}
\end{align*}
\]

\[
(F,\mathcal{E}) \vdash \text{sigexp where type lt = ty} \Rightarrow (N)(\varphi(\mathcal{E}))
\]

For `type`:

\[
B \vdash \text{type } t \Rightarrow (\{tn\}((t \mapsto tn),\{\},\{\})) \text{ (tn fresh)}
\]

\[
\begin{align*}
\mathcal{E} \vdash ty &\Rightarrow \tau \\
\text{tyvars}(\tau) &\Rightarrow \alpha_1 \ldots \alpha_n \\
\sigma &\Rightarrow \forall \alpha_1 \ldots \alpha_n \cdot \tau
\end{align*}
\]

\[
(F,\mathcal{E}) \vdash \text{val } v:\text{ty} \Rightarrow ((\{x\}((\{\},\{v \mapsto \tau\})),\{\}))
\]

\[
B \vdash \text{sigexp } \Rightarrow (N)\mathcal{E}_i
\]

\[
B \vdash \text{structure } s : \text{sigexp } \Rightarrow (N)((\{\},\{\}),\{s \mapsto \mathcal{E}_1\})
\]

\[
B \vdash \text{spec}_1 \Rightarrow (N_1)\mathcal{E}_1 \quad B \vdash \text{spec}_2 \Rightarrow (N_2)\mathcal{E}_2
\]

\[
\text{Dom}(\mathcal{E}_1) \cap \text{Dom}(\mathcal{E}_2) = 0
\]

\[
B \vdash \text{spec}_1 \text{ spec}_2 \Rightarrow (N_1 \cup N_2)(\mathcal{E}_1 + \mathcal{E}_2)
\]

For `strexp`:

\[
B \vdash \text{strdec } \Rightarrow (N,\mathcal{E},d)
\]

\[
B \vdash \text{struct strdec end } \Rightarrow (N,\mathcal{E},d)
\]

\[
\text{lookUp}(ls,\mathcal{E}) = \mathcal{E}_i
\]

\[
(F,\mathcal{E}) \vdash \text{strexp } \Rightarrow (N_1,\mathcal{E}_1,d_1)
\]

\[
F(f) \geq (\mathcal{E}_2 \rightarrow (N_3)(\mathcal{E}_3,d_3))
\]

\[
\mathcal{E}_1 \rightarrow \mathcal{E}_2 \quad (N_2,d_2) = \text{Inst}(\mathcal{E}_1,\mathcal{E}_2)
\]

\[
(F,\mathcal{E}) \vdash f(\text{strexp}) \Rightarrow (N_1 \cup N_2 \cup N_3,\mathcal{E}_3,d_1,d_2,d_3)
\]

\[
B \vdash \text{strexp } \Rightarrow (N_1,\mathcal{E}_1,d)
\]

\[
B \vdash \text{sigexp } \Rightarrow (N_2)\mathcal{E}_2
\]

\[
(N_2\mathcal{E}_2 \geq \mathcal{E}_2 \quad \mathcal{E}_1 \rightarrow \mathcal{E}_2)
\]

\[
B \vdash \text{strexp : sigexp } \Rightarrow (N_1,\mathcal{E}_1',d)
\]

For `strdec`:

\[
(F,\mathcal{E}) \vdash \text{dec } \Rightarrow (N,\mathcal{E},d)
\]

\[
(F,\mathcal{E}) \vdash \text{strexp } \Rightarrow (N,\mathcal{E}_1,d)
\]

\[
(F,\mathcal{E}) \vdash \text{structure } s = \text{strexp } \Rightarrow (N,((\{\},\{\}),\{s \mapsto \mathcal{E}_1\}),d)
\]

\[
(F,\mathcal{E}) \vdash \text{strdec}_1 \Rightarrow (N_1,\mathcal{E}_1,d_1)
\]

\[
(F,\mathcal{E} + \mathcal{E}_1) \vdash \text{strdec}_2 \Rightarrow (N_2,\mathcal{E}_2,d_2)
\]

\[
(F,\mathcal{E}) \vdash \text{strdec}_1 \text{ strdec}_2 \Rightarrow (N_1 \cup N_2,\mathcal{E}_1 + \mathcal{E}_2,d_1,d_2)
\]

For `topdec`:

\[
B \vdash \text{strdec } \Rightarrow (N_1,\mathcal{E}_1,d_1)
\]

\[
B \vdash \text{strdec } \Rightarrow ((\{\},\mathcal{E}_1),d_1)
\]

\[
(F,\mathcal{E}) \vdash \text{sigexp } \Rightarrow (N_1)\mathcal{E}_1
\]

\[
(F,\mathcal{E} + ((\{\},\{s \mapsto \mathcal{E}_1\})) \vdash \text{strexp } \Rightarrow (N_2,\mathcal{E}_2,d_1)
\]

\[
(F,\mathcal{E}) \vdash \text{functor } f(s;\text{sigexp})\text{ strexp } \Rightarrow (\{(f \mapsto (N_1)(\mathcal{E}_1 + (N_2)(\mathcal{E}_2,d_1)))\},\{\}),\{\},\{\})\epsilon)
\]

\[
B \vdash \text{topdec}_1 \Rightarrow (B_1,\mathcal{E}_1)
\]

\[
B + B_1 \vdash \text{topdec}_2 \Rightarrow (B_2,\mathcal{E}_2,d_2)
\]

\[
B \vdash \text{topdec}_1 \text{ topdec}_2 \Rightarrow (B_1 + B_2,\mathcal{E}_1 + \mathcal{E}_2,d_1,d_2)
\]

Fig. 7 Compilation Rules for Modules

and $N$ is the set of declared variables of $x_2$'s. This auxiliary function is used in the functor application compilation for type coercion from the actual types to instantiated formal types for the specified value identifiers in the functor argument.

We note that the compilation algorithm is a refinement of the static interpretation. From this construction, the compilation algorithm yields a semantic object for any type consistent source program. Moreover, the type part of the semantic ob-
ject represents the type information of the program. To formalize this property, we define the term erasure $\overline{X}$ of a semantic object $X$ as follows.

$$
\frac{\overline{F}, \overline{E}}{\overline{F}, \overline{E}} = (\overline{F}, \overline{E})
$$

$$
(F, E) \vdash \text{strexp} \Rightarrow (N_1, E_1, d_1)
$$

$$
F(f) = (B_0, s_0, \text{strexpo}, \Phi)
$$

$$
\Phi \geq (\overline{E_2} \rightarrow (\overline{N_2})\overline{E_3})\ E_1 \succ E_2
$$

$$
B_0 + (\{\}, \{\}, \{\}, \{\}) \vdash \text{strexpo}
$$

$$
\Rightarrow (N_2, E_3, d_2)
$$

This rule says functor application flattens not only the functor actual argument but also the declared functor body strexpo. In contrast, ours directly links the “refreshed” functor body directly. Here “refreshed” reflects two things: One is to instantiate the formal argument with real ones that is modeled by the functor signature substitution $\varphi^+$, while another is to refresh the declared variables that is modeled by “$N_3$ fresh”.

The approach in [8] compiles the functor body at the functor application under the real implementation types, which guarantees the success of various static property computation on the types. However we claim that this computation can be done at linking time with the assumption that all the compilation phases up to the phase that provides the target language for functor application are compatible with this first compilation and then linking strategy. SML\textsc{f} takes the type directed compilation method [13] overall, which is compatible with our proposed functor compilation strategy. Thus by treating functor as a code template without flattening again and again at each functor application site, this is the point we overcome.

## 4 Type Soundness of Compilation

We say that a compilation environment $A$ is well-formed if all the typed variables of the form $(\sigma, x : \sigma')$ satisfies the condition $\sigma' \succ \sigma$ and write $\vdash A$. We omit its trivial inductive definition.

Now we shall show several theorems and corollaries that finally lead to the type soundness theorem.

The following theorem shows that if a structure declaration, a structure expression or a top declaration can be type checked, then it can be translated into the intermediate language. Its proof follows the overall proof structure in [8] with new proofs for functor related cases and is given in the appendix.

**Theorem 1.** Let phrase be a structure declaration, a structure expression, or a top declaration, and $B$ be any well-formed compilation basis. If $\overline{B} \vdash\text{phrase} \Rightarrow E_1$ (or $\overline{B} \vdash\text{phrase} \Rightarrow B_1$), then there are some $E_1(\text{or } B_1)$ and $d$ such that $E_1 = \overline{E_1}$ (or $B_1 = \overline{B_1}$), $B \vdash \text{phrase} \Rightarrow (N, E_1, d)$ (or $B \vdash \text{phrase} \Rightarrow (B_1, d)$), $\vdash E_1$ (or $\vdash B_1$).

By the theorem above, the following corollary directly follows.

**Corollary 1.** Let $B$ be any well formed compilation basis. If $\overline{B} \vdash \text{topdec} \Rightarrow B_1$ then there are some $B_1$ and $d$ such that $B_1 = \overline{B_1}$, and $B \vdash \text{topdec} \Rightarrow (B_1, d)$, $\vdash B_1$. Then we show that if a phrase can be translated into the intermediate language under some compilation basis $B$, then the intermediate language can be typable under the type environment $\Gamma$ that is type consistent with $B$, written $\Gamma \vdash_{\text{te}} B$. 

66
Theorem 2. Let phrase be either a structure declaration, a structure expression, or a top declaration. Moreover, let \( A \) be either a translation environment or a translation basis. If \( B \vdash \text{phrase} \Rightarrow (N, A, d) \) (or \( B \vdash \text{phrase} \Rightarrow (A, d) \)) and \( \Gamma \vdash_{tc} B \) then there exists \( \Gamma' \) such that \( \Gamma \vdash d : \Gamma' \) and \( \Gamma + \Gamma' \vdash_{tc} A \) and \( \text{Dom} \, \Gamma' \subseteq N \).

By the theorem above, the following corollary directly follows.

Corollary 2. If \( B \vdash \text{topdec} \Rightarrow (B_1, d) \) and \( \Gamma \vdash_{tc} B \) then there exists \( \Gamma' \) such that \( \Gamma \vdash d : \Gamma' \) and \( \Gamma + \Gamma' \vdash_{tc} B_1 \).

Now with the combination of Corollary 1, 2 above and the type soundness Theorem 4 in appendix A.3 for the intermediate language proved in [8], we have the type soundness property of the module language topdec with respect to the operational semantics (see appendix A.2) given for the type erased intermediate language \( er(d) \) as follows.

Theorem 3. If \( \xi \vdash \text{topdec} \Rightarrow B \) and \( \Gamma \vdash_{tc} B \) then there exists \( \Gamma' \) such that

- \( B \vdash \text{topdec} \Rightarrow (B', d) \)
- \( \Gamma + \Gamma' \vdash_{tc} B' \)
- \( \Gamma \vdash d : \Gamma' \)
- If there exists \( \varphi \) such that \( D \vdash er(d) \rightsquigarrow \varphi \) then \( \varphi \neq \text{wrong} \).

5 Implementation

We develop a SML\(^\sharp\) compiler for full Standard ML language. SML\(^\sharp\) has many intermediate languages, which thus enjoys the nice property of extending one specific intermediate language for new features. The major phases in the front-end of SML\(^\sharp\) are given in Figure 8. Embedded in the major phases, the minor phases consist of a recursive value declaration optimization phase, a type variable setting phase, an uncurry optimization phase, a print code generation phase, a type checker phase, an inlining phase, an multivalue calculus type checker phase.

Functor linker phase is the final phase for functor compilation and also the final phase of the compiler front-end. This phase passes the intermediate language produced by the front-end to the back-end about which we are not concerned in this paper.

5.1 The Source Language and The Target Language

The source language of module compilation is a type annotated intermediate language produced by the type checking phase. The target flattened language is the output language produced by the functor linker phase. Their definitions are almost straightforward translations from the formal definition. Thus we omit them here.
5.2 Module Compilation

In this part we shall introduce several implementation issues we met in the module compilation by examples.

5.2.1 Unique Variable Allocation

Given a sequence of code, module compiler flattens the source language, that is, module constructs disappear and the long value identifiers are compiled into the unique variables. For example,

```sml
#structure S =
  struct
    val x = 1
  end
val y = S.x;
```

Its compiled result is as follows.

```sml
val $1:int = 1;
val $2:int = $1;
```

where $S.x$ and $y$ respectively obtain the unique variable $\$1$ and $\$2$.

5.2.2 Global Index Allocation

Since SML♯ evaluates codes interactively, it maintains some global arrays to hold global values. Compiler allocates a slot in the global array for each flattened top level declared variable. Each slot stores the value for each corresponding variable. Global indices are used to identify the slots, and thus each variable associates a global index. We illustrate this process by the following example.

```sml
# structure S =
  struct
    fun f x = x
  end
val g = S.f

# functor F(A:sig val f : int -> int end) = struct
val g = A.f
end
# structure K = F(S)
```

Module compiler compiles the code fragment above into the following intermediate language.

```sml
# InitializeGlobalArray(&1,1024,BOXEDty,loc1);
val $1:'a -> 'a = fn x => x;
val $2:'a -> 'a = $1;
SetGlobalValue($1, 0, BOXEDty);
SetGlobalValue($1, 1, $2, BOXEDty);
(* static binding for F *)
InitF
```

A subtle case occurs for the compilation of the value identifiers in the functor body. Consider the following example,

```sml
# val x = 1
val y = x
functor F(A:sig end) = struct
  val z = x
end
```
# structure K = F(struct end)
The value identifier x and the functor F lie in the same compilation unit. Module compiler has to compile x into a global index denoted variable. If the functor F is applied in another sequence of code following a new # prompt as above, that variable in the instantiated code template expects to refer to the value in the global array. The compiled target code is as follows.

```
# InitializeGlobalArray(&1,1024,ATOMty,loc1)
val $1:int = 1
val $2:int = $1
SetGlobalValue(&1, 0, $1, ATOMty)
SetGlobalValue(&1, 1, $2, ATOMty)
(* static binding for F *)
F |->
  ()
  ()
  ->
  (($3})
  {
    {},
    {z -> (int,($3:int))},
    {}
  },
  {val $3:int =
    GetGlobalValue(&1,0,ATOMty,loc1)}
  )

# val $4:int =
  GetGlobalValue(&1, 0, int, loc2)
SetGlobalValue(&1, 2, $4, ATOMty)
```

5.2.3 Type Issues

### Structure Curtailment

A signature constraint curtails the structure. Module compiler treats this similarly to what the type checker does. Consider the following example,

```
structure A =
  struct
    val y = 1.6
  end
structure A =
  struct
    val x = 1
    val y = true
  end : sig
```

Without consideration of the signature constraint, the latter A.y overrides the former one. Thus the environment instance representing the second bare structure A as follows

```
({},
 {x -> (int,($1:int))},
 y -> (bool,($2:bool))
}()
```

is cut down into

```
({}, {x -> (int,($1:int))}, {})
```

by only exposing x to outside.

### Implementation Type Propagation

SML♯ depends on the implementation type to compute their various static properties that are used in some powerful extensions of SML♯ [14][15] and its interoperability feature [19]. Thus the opaque type should carry the implementation for inspection and then is defined to be consisting of an abstract type and an implementation type. In the following example,

```
structure S =
  struct
    type t = int
  end : sig type t end
```

Type S.t is internally represented as (S.t, int) where the first component is for type checking and the second one is for static computation.

### Type Directed Compilation

SML♯ adopts the type directed compilation strategy. If a structure is constrained by a signature with narrower type specified for value identifiers, the more concrete implementation type should be passed on to the declared function. For example, the following structures,

```
structure S1 = struct fun f x = x end
structure S2 = S1:sig val f:int -> int end
```

are compiled into

```
val $1:'a -> 'a = Λα.λ$2:α.x
val $3:int -> int = λ$4:int.($1 {int} $4)
```
Abstract Type Instantiation

Similar to that of opaque signature constraint, abstract types specified in the functor argument should be instantiated into implementation types at the functor application spot. Consider the following example,

```sml
# structure A =
  struct
    val x = 1
    type t = int
  end

# functor F(S: sig
    type t
    val x : t
  end) =
  struct
    val y = S.x : S.t
    datatype s = foo of t
    val z = (foo x) : s
  end

# structure B = F(A)
```

The module compiler assigns a unique type name to each abstract type specified in functor argument. Suppose we have the following mapping \{t \mapsto \#1\}. At functor application, SML\# compiler generates a mapping from the type name to the implementation type by computing the formal functor argument environment and the actual one. In this case, the actual argument structure A stores the implementation type for t, which is a mapping \{t \mapsto \text{int}\}. The functor application F(A) drives the compiler to generate a substitution \{\#1 \mapsto \text{int}\} that is used to instantiate all those references to the abstract type t in the functor body environment and code template for the functor F.

In addition to the abstract types, those datatypes declared in the functor body need to be refreshed to reflect the instantiation of the abstract type t. Thus a mapping from the type name of s to its instantiated implementation type is constructed. By the combination of type substitution constructed for functor argument and functor body, the types S.t, s in code template are all instantiated.

### 5.2.4 Datatype and Exception

As a full Standard ML language consistent compiler, SML\# also needs to treat datatype declaration, exception declaration that are missing in our formal presentation.

For both cases, no runtime code is produced at their definition and instead the constructors are expanded at their usage spot in the type checking phase. Each data constructor attains a relative, that is, not global, unique tag with respect to the data constructors belonging to the same type. The underlying reason behind this strategy is that the type checking guarantees all the constructors matching the value of some expression should be of the same type. Hence the global uniqueness enforcement is not necessary. As for the exception constructors, since all of them belong to the same exception type, the notion of “global” applies here with respect to all the exception constructors.

We use datatype to illustrate. Exception cases are quite similar. For example,

```sml
datatype t = foo of int
val x = foo 1
```

The type checking of the datatype declaration only produces a static binding \{t \mapsto \text{(\#1, foo: int -> t)}\}. At the data constructor application spot, foo is expanded into a function as follows

```sml
TPFNM
{
  (* function arguments *)
  argVarList = [(x, int)]
  (* function return type *)
  bodyTy = t,
  (* function body *)
  bodyExp=
    TPCONSTRUCT
    {
      (* constructor information *)
      con = {name = foo,
             ty = int -> t,
             tag = 1},
      (* nil for monomorphic function type *)
      instTyList = nil,
      (* reference to function argument *)
      argExpOpt =
        SOME (TPVAR (
                  {name = x,
                   path = NilPath,
                   ty = int}),
               loc),
      loc=loc
    },
    loc=loc
}
```

and thus constructor application is compiled into
function application. Module compilation phase does not touch the data constructors and only translates them into another intermediate language constructs with direct correspondence. Our match compiler takes the responsibility of the pattern matching compilation.

6 Benchmark

One of our purpose is to avoid the non-trivial compilation time introduced by the same functor applied many times. To show this, we slightly adapt mlyacc and barnes_hut benchmarks in SML/NJ [4] benchmark suite for multiple functor applications. The purpose of the changes is to filter out the compilation time for other language constructs other than functor applications. Thus we name them mlyacc∗ and barnes_hut∗ to distinguish them from the original version. For the former, we put all the top level functor applications as a functor body of one functor with empty formal argument, and let it be the top entry functor and apply it several times. For the latter, we simply apply the top entry “Main” functor several times. Table 1 shows the result with different application times \( (n = 1, 3, 5) \) and the incremental time (+). The incremental time exactly corresponds to the time cost for the compilation of the functor applications in these two compilers.

Our environment is IBM ThinkPad T43p with VMWare WorkStation 5.0/Mandrake 9.0. The compilers are SMLζ current working version and MLKit 4.1.4.

7 Conclusion

We have presented an approach to compile the SML module language into an intermediate language without module constructs which is shared with the SML core language. Especially the functor is compiled in the way that it is regarded as a code template, and instantiated at each functor application. Module flattening does not incur any extra complexity, and in this way the optimization and program analysis based on the intermediate language can serve both language. On the other hand, SMLζ is trying to achieve the cut-off separate compilation. In the cut-off separate compilation framework, functor body will be compiled into native object code instead of the intermediate language. Functor application exactly imitates the ordinary linking process. Our “functor is code template” compilation method takes a promising step towards this direction. Currently the proposed module compilation method has been fully implemented in our SMLζ compiler and extends to the whole Standard ML language specified in The Definition of Standard ML [6]. Our future cut-off separate compilation framework will take advantage of this proposed flattening strategy.

Table 1 Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>SMLζ Compilation Time</th>
<th>MLKit 4.1.4 Compilation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n=1</td>
<td>n=3</td>
</tr>
<tr>
<td>mlyacc∗</td>
<td>13.13s</td>
<td>31.44s</td>
</tr>
<tr>
<td>barnes_hut∗</td>
<td>2.42s</td>
<td>4.03s</td>
</tr>
</tbody>
</table>

References

[13] Ohori, A.: Type-Directed Specialization of


A Typing Rules and Dynamic Semantics of The Intermediate Language

A.1 Typing Rules

Typing environment $\Gamma \in Var \xrightarrow{fin} TypeScheme$
is a mapping from unique variables to their type schemes.

$$
\begin{align*}
\Gamma + \{ x \mapsto \tau \} \vdash e : \tau' & \quad x \notin \text{Dom}(\Gamma) \\
\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau' & \\
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash e_1 e_2 : \tau_2 & \\
\Gamma(x) \triangleright \tau & \\
\Gamma \vdash x : \tau & \\
\Gamma \vdash \text{let } d \text{ in } e : \tau & \\
x \notin \text{Dom}(\Gamma) & \quad \Gamma \vdash e : \tau \\
\alpha_1 \ldots \alpha_n \notin \text{tyvars}(\Gamma) & \\
\Gamma \vdash \text{val } x : (\alpha_1, \ldots, \alpha_n).\tau = e & \\
: \{ x \mapsto \forall (\alpha_1, \ldots, \alpha_n).\tau \} & \\
\Gamma \vdash d_1 : \Gamma_1 & \quad \Gamma + \Gamma_1 \vdash d_2 : \Gamma_2 \\
\Gamma \vdash d_1 d_2 : \Gamma_1 + \Gamma_2 & \\
\Gamma \vdash e : \{ \}
\end{align*}
$$

A.2 Dynamic Semantics

Type erasure function $er$ is defined to transform the typed expression and declaration into untyped ones.

$$
\begin{align*}
er(\lambda x : \tau. e) & = \lambda x. er(e) \\
er(e_1 e_2) & = er(e_1) \ er(e_2) \\
er(x) & = x \\
er(\text{let } d \text{ in } e) & = \text{let } er(d) \text{ in } er(e) \\
er(\text{val } x : \alpha_1, \ldots, \alpha_n. \tau = e) & = \text{val } x = er(e) \\
er(d_1 d_2) & = er(d_1) \ er(d_2) \\
er(\epsilon) & = \epsilon
\end{align*}
$$

Below defines the dynamic objects.

$$
\begin{align*}
v & \in \text{Value} = \text{Closure} \\
\langle \lambda x. e, D \rangle & \in \text{Closure} = Var \times \text{Exp} \times \text{DynEnv} \\
D & \in \text{DynEnv} = Var \xrightarrow{fin} \text{Value} \\
r & \in ExpResult = \text{Value} \cup \{ \text{wrong} \} \\
\emptyset & \in \text{DecResult} = \text{DynEnv} \cup \{ \text{wrong} \}
\end{align*}
$$

Here $Exp$ represents a set of intermediate expressions ranged over by $e$.

$$
\begin{align*}
D \vdash \lambda x. e \rightsquigarrow \langle \lambda x. e, D \rangle & \quad D(x) = v \quad x \notin \text{Dom } D \\
D \vdash e_1 \rightsquigarrow \langle \lambda x. e, D_0 \rangle & \quad D \vdash x \rightsquigarrow \text{wrong} \\
D \vdash e_1 \rightsquigarrow \text{wrong} \\
D \vdash e_1 e_2 \rightsquigarrow \text{wrong} \\
D \vdash e_1 \rightsquigarrow \text{wrong} \\
D \vdash e_1 e_2 \rightsquigarrow \text{wrong} \\
D \vdash d \rightsquigarrow D' & \quad D + D' \vdash e \rightsquigarrow r \\
D \vdash \text{let } d \text{ in } e \rightsquigarrow r \\
D \vdash d \rightsquigarrow \text{wrong} \\
D \vdash \text{let } d \text{ in } e \rightsquigarrow \text{wrong} \\
D \vdash e \rightsquigarrow v \\
D \vdash e \rightsquigarrow \text{wrong} \\
D \vdash \text{val } x = e \rightsquigarrow \{ x \mapsto v \} \\
D \vdash e \rightsquigarrow \text{wrong} \\
D \vdash \text{val } x = e \rightsquigarrow \text{wrong} \\
D \vdash d_1 \rightsquigarrow D_1 & \quad D + D_1 + d_2 \rightsquigarrow D_2 \\
D \vdash d_1 d_2 \rightsquigarrow D_1 + D_2 \\
D \vdash d_1 \rightsquigarrow \text{wrong} \\
D \vdash d_1 d_2 \rightsquigarrow \text{wrong} \\
D \vdash d_1 \rightsquigarrow \text{wrong} \\
D \vdash d_1 d_2 \rightsquigarrow \text{wrong} \\
D \vdash d_1 \rightsquigarrow \{ \}
\end{align*}
$$

72
A.3 Type Soundness

Theorem 4. If $\Gamma \vdash d : \Gamma'$ and $D \vdash er(d) : \varphi$, then $\varphi \neq \text{wrong}$.

Proof. Given in [8].

B Lemmas and Theorems

Lemma 1. Let phrase be specification expression or signature expression. Moreover, let $A$ be a typing environment and let $A$ be a compilation environment. If $\Gamma \vdash phrase \Rightarrow A$ and $B$ is well formed then there exists $A$ such that $\Gamma \vdash phrase \Rightarrow A$ and $\vdash A$ and $\overline{A} = A$.

Proof. The proof is by induction on the structure of spec, sigexp.

- case spec = val $v : ty$. From the assumptions and typing rule for value specification, we have $\overline{\text{E}} \vdash \text{spec} \Rightarrow (\{v \mapsto \sigma\}, \{\})$

Let $\mathcal{E} = (\{v \mapsto (\sigma, x : \sigma)\}, \{\})$, $x$ is fresh, then we have there exists $\mathcal{E}$ such that $\overline{\text{B}} \vdash \text{val} \ v : \sigma \Rightarrow (\{x\}\mathcal{E})$

By I.H., we have $\overline{\text{B}} \vdash \text{spec} \Rightarrow (T)E$

By I.H., we have $\overline{\text{B}} \vdash \text{spec} \Rightarrow (N)\mathcal{E}$

By the definition of erasure, we have $\overline{\mathcal{E}_1 + \mathcal{E}_2} = E_1 + E_2$

Then by the compilation rule, we have $\overline{\text{B}} \vdash \text{spec} \Rightarrow (\mathcal{E}_1 + \mathcal{E}_2)$

as required.

- case sigexp = sig spec end. It directly follows by I.H.

- case sigexp = sigexp′ where type $lt \ = \ ty$. From the assumption, it follows that $\overline{(F, E) \vdash \text{sigexp}} \Rightarrow (T)E_1$

By the definition of erasure, we have $\overline{\mathcal{E}_1 + \mathcal{E}_2} = E_1 + E_2$

Then by the compilation rule, we have $\overline{\text{B}} \vdash \text{spec} \Rightarrow (\mathcal{E}_1 + \mathcal{E}_2)$

as required.

Theorem 1. Let phrase be a structure declaration, a structure expression, or a top declaration, and $B$ be any well formed compilation basis. If $\overline{\text{B}} \vdash \text{phrase} \Rightarrow E_1$ (or $\overline{\text{B}} \vdash \text{phrase} \Rightarrow B_1$), then there are some $E_1$ (or $B_1$) and $d$ such that $E_1 = \overline{\mathcal{E}_1}$ (or $B_1 = \overline{\mathcal{E}_1}$), $\overline{\text{B}} \vdash \text{phrase} \Rightarrow (N, \mathcal{E}_1, d)$ (or $\overline{\text{B}} \vdash \text{phrase} \Rightarrow (B_1, d)$), $\vdash \mathcal{E}_1$ (or $\vdash \mathcal{E}_1$).

Proof. The proof is by induction over the structure of strdec, strexp, and topdec. We only prove the cases that are not covered in [8].

- case strexp = $f(strexp')$. Let $\overline{\text{B}} = (F, E)$.

From the assumptions, we have $\overline{\text{B}} \vdash strexp' \Rightarrow E$

By the definition of erasure, we have $\overline{\text{E}_1 + \text{E}_2} = \text{E}_1 + \text{E}_2$

Thus it follows that $\overline{\text{B}} \vdash \text{spec} \Rightarrow (N)(\varphi(E))$
By I.H., we get
\[ B \vdash \text{strexpr} \Rightarrow (N_1, \mathcal{E}, d) \]
\[ \vdash \mathcal{E} \]
\[ \varepsilon = E \]

Let \( F(f) = (T_1)(E_1 \rightarrow E_2) \) and \( \phi \) be the substitution such that \( \phi(E_1 \rightarrow E_2) = E' \rightarrow E'' \).

Let \( F(f) = (N_2)(\mathcal{E}_2 \rightarrow N_3(\mathcal{E}_3, d_2)) \). From the assumption, we have \( \neg \mathcal{E}_2 = T_1, \mathcal{E}_3 = E_2 \).

Let \( \phi \subseteq \mathcal{E}^+ \) and \( \mathcal{E}^+ \backslash \phi \) be the substitution from variable names in \( N_2 \) to the corresponding variable names in \( \mathcal{E} \). Let \( \mathcal{E}^+(\mathcal{E}_2 \rightarrow N_3(\mathcal{E}_3, d_2)) = (\mathcal{E}_4 \rightarrow N_5(\mathcal{E}_5, d_3)) \) and \( N_5 \) be fresh. Let \( (N_6, d_1) = \text{Inst}(\mathcal{E}_4, \mathcal{E}_5) \). Therefore we have
\[ B \vdash f(\text{strexpr}) \Rightarrow (N_1 \cup N_6 \cup N_5, \mathcal{E}_5, d_1, d_3) \]
as required. By \( \mathcal{E}_5 = E_2 \) and \( \mathcal{E}_5 \) is a alpha-renaming of bound variable names of \( \mathcal{E}_3 \), thus \( E_3 = E_2 \).

From the assumption \( \vdash \mathcal{E}_3 \), and from the construction of \( \mathcal{E}^+ \), thus \( \varepsilon \).

- case topdec = function \( f(s, \text{sigexpr}) = \text{strexpr} \).
  From the assumptions, we have
  \[ B \vdash \text{sigexpr} \Rightarrow (T_1)E_1 \]
  \[ B + \{\{\}, \{\}, \{s \mapsto E_1\}\} \vdash \text{strexpr} \Rightarrow E_2 \]
  \[ F = \{f \mapsto (T_1)(E_1 \rightarrow E_2)\} \]
  \[ B \vdash \text{topdec} \Rightarrow (F, \{\}) \]
  \[ B \]

By Lemma 1, we have
\[ B \vdash \text{sigexpr} \Rightarrow (N_1)\mathcal{E}_1 \]
\[ \vdash \mathcal{E}_1 \]
\[ (N_1)\mathcal{E}_1 = (T_1)E_1 \]

Then by the definition of erasure we have
\[ B + \{\{\}, \{\}, \{s \mapsto \mathcal{E}_1\}\} \vdash \text{strexpr} \Rightarrow E_2 \]

By the definition of well-formedness, we have
\[ \vdash B + \{\{\}, \{\}, \{s \mapsto \mathcal{E}_1\}\} \]

By I.H., we have
\[ B + \{\{\}, \{\}, \{s \mapsto \mathcal{E}_1\}\} \vdash \text{strexpr} \Rightarrow (N_2, \mathcal{E}_2, d) \]
\[ \vdash \mathcal{E}_2 \]
\[ \mathcal{E}_2 = E_2 \]

Let \( F \) be the compilation functor environment
\[ F = \{f \mapsto (N_1)(\mathcal{E}_1 \rightarrow (N_2)(\mathcal{E}_2, d))\} \]

By the definition of erasure,
\[ (F, \{\}) = (F, \{\}) \]
By the definition of well-formedness,
\[ \vdash (F, \{\}) \]
Thus we have
\[ B \vdash \text{topdec} \Rightarrow (F, \{\}, \varepsilon) \]
as required.

Lemma 2. If \( \Gamma \vdash e : \tau, \Gamma \vdash_{tc} \Gamma' \), then \( \Gamma + \Gamma' \vdash e : \tau \) and \( \Gamma'' + \Gamma' \vdash e : \tau \).

Proof. Do induction on the derivation and do case analysis on the last rule.

- case \( e = \lambda x : \tau. e \). It follows that \( \Gamma + \{x \mapsto \tau\} \vdash e : \tau' \), \( x \notin \text{Dom}(\Gamma) \). By alpha conversion, we can require \( x \notin \text{Dom}(\Gamma') \), thus by \( \Gamma \vdash_{tc} \Gamma' \), we have \( \Gamma + \{x \mapsto \tau\} \vdash_{tc} \Gamma' + \{x \mapsto \tau\} \). By I.H., we have \( \Gamma + \Gamma' \vdash \lambda x : \tau.e : \tau \rightarrow \tau' \) and \( \Gamma' + \Gamma \vdash \lambda x : \tau.e : \tau' \rightarrow \tau \).
- case \( e = x \). It follows that \( \Gamma(x) \vdash \tau \). Since \( \Gamma \vdash_{tc} \Gamma' \), we have \( \Gamma(x) = \Gamma'(x) = \Gamma + \Gamma \vdash x : \tau \) and \( \Gamma' + \Gamma \vdash x : \tau \).

The other cases follow from I.H. directly.

Lemma 3. If \( \Gamma \vdash d : \Gamma' \), then \( \text{Dom}(\Gamma') \cap \text{Dom}(\Gamma) = \emptyset \) and \( \text{decl}(d) = \text{Dom}(\Gamma') \).

Proof. Given in [8].

Lemma 4. If \( \Gamma \vdash d : \Gamma' \), \( \Gamma \vdash_{tc} \Gamma'' \), then \( \Gamma + \Gamma'' \vdash d : \Gamma' \) and \( \Gamma'' + \Gamma \vdash d : \Gamma' \).

Proof. Do induction on the derivation and do case analysis on the last rule.

- case \( d = \text{val } x : \alpha_1 \ldots \alpha_n. e = c \). It follows that \( x \notin \text{Dom}(\Gamma), \alpha_1 \ldots \alpha_n \notin \text{tyvars}(\Gamma) \), \( \Gamma \vdash e : \tau \).

By alpha conversion and Lemma 3, we can always require \( \text{Dom}(\Gamma') \cap \text{Dom}(\Gamma'') = \emptyset \), and then we have \( x \notin \text{Dom}(\Gamma'') \). Thus we have \( x \notin \text{Dom}(\Gamma' + \Gamma'') \) and \( x \notin \text{Dom}(\Gamma'' + \Gamma) \). By type variable alpha-renaming, we have \( \alpha_1 \ldots \alpha_n \notin \text{tyvars}(\Gamma' + \Gamma'') \) and \( \alpha_1 \ldots \alpha_n \notin \text{tyvars}(\Gamma'' + \Gamma) \). By Lemma 2, we have \( \Gamma + \Gamma'' \vdash e : \tau \) and \( \Gamma'' + \Gamma \vdash e : \tau \). Thus we have \( \Gamma + \Gamma'' \vdash \text{val } x : \sigma = e : \Gamma' \) and \( \Gamma'' + \Gamma \vdash \text{val } x : \sigma = e : \Gamma'' \).

The other cases follow from I.H. directly.

Lemma 5. If \( \Gamma_1 \vdash_{tc} \mathcal{E} \) and \( \text{Dom}(\Gamma_1) \cap \text{Dom}(\Gamma_2) = \emptyset \) then \( \Gamma_1 + \Gamma_2 \vdash_{tc} \mathcal{E} \).

Proof. Given in [8].

Lemma 6. If \( \Gamma_1 \vdash_{tc} \mathcal{E} \) and \( \Gamma_1 \vdash_{tc} \Gamma_2 \), then
\( \Gamma_1 + \Gamma_2 \vdash_{tc} \mathcal{E} \) and \( \Gamma_2 + \Gamma_1 \vdash_{tc} \mathcal{E} \).

**Proof.** Easily adapted from the proof for Lemma 5.

For the base case, it follows from \( \Gamma_1 \vdash_{tc} \Gamma_2 \) that \( \Gamma_1(x) = (\Gamma_1 + \Gamma_2)(x) = (\Gamma_2 + \Gamma_1)(x) \) for all \( x \in Dom(\Gamma_1) \cap Dom(\Gamma_2) \). Then it is proved. \( \square \)

**Lemma 7.** If \( \Gamma_1 \vdash_{tc} \Gamma_2 \) and \( \Gamma' \vdash_{tc} \Gamma_1 + \Gamma_2 \) then \( \Gamma_2 \vdash_{tc} \Gamma_1 + \Gamma' \).

**Proof.** Suppose a generic \( x \) such that \( x \in (Dom(\Gamma_2) \cap Dom(\Gamma_1 + \Gamma')) \).

- case \( x \in \text{Dom}(\Gamma_1) \) and \( x \notin \text{Dom}(\Gamma') \). From \( \Gamma_1 \vdash_{tc} \Gamma_2 \) we have \( \Gamma_2(x) = \Gamma_1(x) = (\Gamma_1 + \Gamma')(x) \).
- case \( x \in \text{Dom}(\Gamma_1) \) and \( x \notin \text{Dom}(\Gamma') \). From \( \Gamma' \vdash_{tc} \Gamma_1 + \Gamma_2 \) and \( x \in \text{Dom}(\Gamma_2) \), we have \( \Gamma'(x) = (\Gamma_1 + \Gamma_2)(x) = \Gamma_2(x) \).
- case \( x \notin \text{Dom}(\Gamma_1) \) and \( x \in \text{Dom}(\Gamma') \). Similar as the above case. \( \square \)

**Lemma 8.** If \( \Gamma_1 \vdash B \) and \( \Gamma_1 \vdash_{tc} \Gamma_2 \) then \( \Gamma_1 + \Gamma_2 \vdash B \).

**Proof.** Let \( B \) be \((\mathcal{F}, \mathcal{E})\).

From the assumptions we have
\[
\Gamma_1 \vdash_{tc} \mathcal{E} \\
\Gamma_1 \vdash_{tc} \Gamma_2
\]
From Lemma 6, we have
\[
\Gamma_1 + \Gamma_2 \vdash_{tc} \mathcal{E}
\]
Consider a generic \( C \) such that
\[
C \in \text{Ran}(\mathcal{F})
\]
\[
C \geq (\mathcal{E}_1 \rightarrow (N_2)(\mathcal{E}_2, d))
\]
\( \text{N}_2 \) fresh
Consider a generic \( \Gamma' \) such that
\[
\Gamma' \vdash_{tc} \mathcal{E}_1 \\
\Gamma' \vdash_{tc} \Gamma_1 + \Gamma_2
\]
Thus from \( \Gamma_1 \vdash_{tc} \Gamma_2 \) and the definition of type consistency, we have
\[
\Gamma' \vdash_{tc} \Gamma_1
\]
From the definition of type consistency, we have there exists \( \Gamma'' \) such that
\[
\Gamma_1 + \Gamma' \vdash d : \Gamma'' \\
\Gamma_1 + \Gamma' + \Gamma'' \vdash_{tc} \mathcal{E}_2
\]
\( \text{Dom}(\Gamma'') \subseteq N_2 \)
By Lemma 7, we have
\[
\Gamma_2 \vdash_{tc} \Gamma_1 + \Gamma'
\]
From Lemma 4, we have
\[
\Gamma_1 + \Gamma' + \Gamma_2 \vdash d : \Gamma''
\]
Due to the type consistency, it can be easily ver-

ified that
\[
\Gamma_1 + \Gamma' + \Gamma_2 = \Gamma_1 + \Gamma_2 + \Gamma'
\]
Thus we have
\[
\Gamma_1 + \Gamma_2 + \Gamma' \vdash d : \Gamma''
\]
By alpha-renaming of \( N_2 \), we can assume
\[
N_2 \cap \text{Dom}(\Gamma_2) = \emptyset
\]
From Lemma 6 and \( \text{dom}(\Gamma'') \subseteq N_2 \), we have
\[
\Gamma_1 + \Gamma' + \Gamma'' \vdash_{tc} \mathcal{E}_2
\]
By the definition of type consistency again, we have
\[
\Gamma_1 + \Gamma_2 + \Gamma' + \Gamma'' \vdash_{tc} \mathcal{E}_2
\]
\( \square \)

**Lemma 9.** If \( \Gamma_1 \vdash_{tc} B_1 \) and \( \Gamma_1 + \Gamma_2 \vdash_{tc} B_2 \) then \( \Gamma_1 + \Gamma_2 \vdash_{tc} B_1 + B_2 \).

**Proof.** From assumptions we have
\[
\Gamma_1 \vdash_{tc} B_1 \quad (1) \\
\Gamma_1 \vdash_{tc} \Gamma_2 \quad (2)
\]
From Lemma 8, (1) and (2) we have
\[
\Gamma_1 + \Gamma_2 \vdash B_1 \quad (3)
\]
From assumptions we have
\[
\Gamma_1 + \Gamma_2 \vdash B_2 \quad (4)
\]
From the definition of type consistency, (3) and (4), we have
\[
\Gamma_1 + \Gamma_2 \vdash B_1 + B_2
\]
\( \square \)

**Lemma 10.** If \( \Gamma \vdash d : \Gamma' \), then \( \Gamma \vdash_{tc} \Gamma' \).

**Proof.** From Lemma 3, we have \( \Gamma \vdash d : \Gamma' \) implies \( \text{Dom}(\Gamma) \cap \text{Dom}(\Gamma') = \emptyset \), which directly implies \( \Gamma \vdash_{tc} \Gamma' \) from the definition of type consistency. \( \square \)

**Theorem 2.** Let phrase be either a structure declaration, a structure expression, or a top declaration. Moreover, let \( \mathcal{A} \) be either a translation environment or a translation basis. If \( B \vdash \text{phrase} \Rightarrow (N, A, d) \) or \( B \vdash \text{phrase} \Rightarrow (A, d) \) then there exists \( \Gamma' \) such that \( \Gamma \vdash d : \Gamma' \) and \( \Gamma + \Gamma' \vdash_{tc} \mathcal{A} \) and \( \text{Dom} \Gamma' \subseteq N \).

**Proof.** The proof is by induction over the structure of \text{strdec}, \text{strexp} and \text{topdec}. We only prove the cases that are not covered in [8].

- case \( \text{strexp} = f(strexp') \). From assumptions and by the compilation rule, we have
\[
B \vdash \text{strexp'} \Rightarrow (N_1, \mathcal{E}_1, d_1)
\]
\[
B(f) \geq (E_2 \rightarrow (N_2)(E_3, d_3))
\]
\[
\mathcal{E}_2 \geq \mathcal{E}_3
\]
\[
\text{Inst} (\mathcal{E}_1, \mathcal{E}_2) = (N_3, d_4)
\]
\[ \Gamma \vdash tc \ \mathcal{B} \]

\[ N_2, N_3 \text{ fresh} \]

\[ \mathcal{B} \vdash f(strexp) \Rightarrow (N_1 \cup N_2 \cup N_3, \mathcal{E}_3, d_1, d_2, d_3) \]

By I.H., we have

\[ \Gamma \vdash d_1 : \Gamma' \]

\[ \Gamma + \Gamma' \vdash tc \ \mathcal{E}_1 \]

\[ \text{Dom}(\Gamma') \subseteq N_1 \]

By Lemma 10, we have

\[ \Gamma + \Gamma' \vdash tc \ \mathcal{E}_2 \]

\[ \text{Dom}(\Gamma'') \subseteq N_2 \]

By the definition of type consistency for functor, we have

\[ \Gamma + \Gamma' + \Gamma'' \vdash d_3 : \Gamma''' \ i.e. \]

\[ \Gamma + \Gamma' + \Gamma'' + d_3 : \Gamma''' \]

\[ \Gamma + \Gamma' + \Gamma'' + \Gamma''' \vdash tc \ \mathcal{E}_3 \]

\[ \text{Dom}(\Gamma'') \subseteq N_1 \]

By Lemma 10, we have

\[ \Gamma + \Gamma' \vdash tc \ \mathcal{E}_2 \]

\[ \text{Dom}(\Gamma'') \subseteq N_2 \]

By the intermediate language typing rule (A.1), we have

\[ \Gamma \vdash d_1, d_2, d_3 : \Gamma' + \Gamma'' + \Gamma''' \]

\[ \text{Dom}(\Gamma' + \Gamma'' + \Gamma''') \subseteq (N_1 \cup N_2 \cup N_3) \]

- case $\text{topdec} = \text{functor} \ f(s: \text{sigexp}) = \text{strexp}$.

From the assumptions, we have

\[ \mathcal{B} \vdash \text{sigexp} \Rightarrow (N_1) \mathcal{E}_1 \]

\[ \mathcal{B} + (\{\}, \{\}, \{\}, \{s \mapsto \mathcal{E}_1\}) \vdash \text{strexp} \]

\[ \Rightarrow (N_2, \mathcal{E}_2, d) \]

\[ \Gamma \vdash tc \ \mathcal{B} \]

For any $\varphi^+$ such that $\text{Supp}(\varphi^+) \subseteq N_1$, since $\varphi^+$ does not touch names outside $N_1$, it can be easily verified that

\[ \mathcal{B} + (\{\}, \{\}, \{\}, \{s \mapsto \varphi^+(\mathcal{E}_1)\}) \vdash \text{strexp} \Rightarrow \varphi^+(N_2, \mathcal{E}_2, d) \]

Suppose a generic $\Gamma'$ such that

\[ \Gamma' \vdash tc \ \varphi^+(\mathcal{E}_1) \]

\[ \Gamma' \vdash tc \ \mathcal{E}_2 \]

From the definition of type consistency and Lemma 6, we have

\[ \Gamma + \Gamma' \vdash tc \ {s \mapsto \varphi^+(\mathcal{E}_1)} \]

Thus from Lemma 9, we have

\[ \Gamma + \Gamma' \vdash tc \ \mathcal{B} + (\{\}, \{\}, \{\}, \{s \mapsto \varphi^+(\mathcal{E}_1)\}) \]

By I.H., we have

\[ \Gamma + \Gamma' \vdash \varphi^+(d) : \Gamma'' \]

\[ \Gamma + \Gamma' + \Gamma'' \vdash \varphi^+(\mathcal{E}_2) \]

\[ \text{Dom}(\Gamma') \subseteq N_2 \]

Let $\mathcal{F} = \{f \mapsto (N_1)(\mathcal{E}_1 \rightarrow (N_2)(\mathcal{E}_2, d))\}$, thus by the definition of type consistency for functor we have

\[ \Gamma \vdash tc \ (\mathcal{F}, \{\}) \]

By the intermediate language typing rule (A.1), we have

\[ \Gamma \vdash \epsilon : \{\} \]

\[ \square \]