A Separate Compilation Extension to Standard ML (Working Draft)

David Swasey Tom Murphy VII Karl Crary
Robert Harper

January 30, 2006
CMU-CS-06-104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This is a proposal for an extension to the Standard ML programming language to support separate compilation. The extension allows the programmer to write a program broken into multiple fragments in a way that would be compatible between different implementations. It also allows for the separate compilation of these fragments, for incremental recompilation strategies such as cut-off recompilation, and for a range of implementation strategies including whole-program compilation. The semantics of separate compilation is defined independent of the underlying semantic framework for Standard ML and is realized in two forms corresponding to *The Definition of Standard ML* and *The Typed Semantics of Standard ML*.

This material is based on work supported in part by the National Science Foundation under grant 0121633 *Language Technology for Trustless Software Dissemination* and by the Defense Advanced Research Projects Agency under contracts F196268-95-C-0050 *The Fox Project: Advanced Languages for Systems Software* and F196228-91-C-0168 *The Fox Project: Advanced Development of Systems Software*. Any opinions, findings, conclusions and recommendations in this publication are the authors' and do not reflect the views of these agencies.

Keywords: ML, separate compilation

1 Introduction

We propose an extension of Standard ML (SML) to support separate compilation. A separately compiled program fragment, called a *unit*, consists of a series of top-level declarations. A unit is described by an *interface*, which is a series of top-level specifications giving the types of the components of that unit. A unit or interface may make reference to the components of another unit by opening the referenced unit for its use, referring to these components by name. Unit references are *definite*—that is, they refer to specific units rather than abstract arguments—so no sharing specifications are induced by separate compilation [HP05].

An *assembly* is an independently meaningful, yet possibly incomplete, collection of units and interfaces; see Figure 1 for an example. In order to be independently meaningful, an assembly specifies an interface for any externally defined unit to which it refers, and, as a result, it may be compiled independently of them. A unit declaration within an assembly may or may not specify an interface for that unit. If one is specified, the compiled unit is coerced, by a process analogous to signature matching, to the specified interface, which governs all uses of that unit identifier. If no interface is specified, the inferred interface obtained by compiling that unit is used for that unit identifier.[1] By confining attention to a single assembly with no external references, we may support integrated compilation of source code, but we expect that libraries will be organized as assemblies that are compiled separately from and linked against the applications that use them.

A *link script* specifies how to coalesce a series of assemblies into a single assembly, resolving external references in the process. An assembly is *complete*, and therefore eligible to be turned into an executable, when all external references have been resolved. The linker insists that all external references to a given assembly be governed by the same interface, up to a natural extension of signature equivalence to interfaces. The assembly in Figure 1 is incomplete; it can be completed by linking it with an an assembly providing an implementation of the unit `Q` with interface `QUEUE`.

The order of assemblies in a link script is significant; any effects incurred by execution of an assembly occur in the order specified. In particular, there is no conventional "main" entry point, but rather execution begins with the first unit in the completed assembly. A link script may select a subset of the units in an assembly to be retained, along with those units on which they depend. The effects of any omitted units are likewise omitted from the resulting executable. This mode of usage is common for building application code; for libraries it is more typical to include all units in an assembly, regardless of whether they appear to be necessary according to the visible dependencies among them.

An example, illustrating the linking of a few simple assemblies, is given in Figure 2. The labels on the dashed arrows constitute the link script, which determines the order in which linking occurs. In this example the effects of unit `B` precede those of unit `D` because of their order of occurrence in assembly 2. Similarly, the effects of unit `C` precede both of those in assembly 5 because assembly 1 precedes assembly 2 in the link script.

1.1 Key Elements

We give a rigorous semantics of the proposed separate compilation facility in a form that is largely independent of the underlying semantic framework for Standard ML itself. This is achieved by giving the semantics in terms of a collection of *stubs* that provide a narrow, well-specified portal

[1] Inferred interfaces cannot always be written as source interfaces. For example, an interface can be inferred for the declaration `local datatype t = A in val x = A end` but there is no source signature or interface that accurately describes it.

```
interface QUEUE = open (* no opens *) in
    structure Queue :
    sig
        type 'a queue
        val empty : 'a queue
        val push : 'a * 'a queue -> 'a queue
    end
end

unit Q : QUEUE

unit C = open Q in
    val q = Queue.empty
    val q' = Queue.push (0, q)
end
```

Figure 1: A simple assembly. The interface `QUEUE` describes units that declare a structure `Queue`. The assembly requires unit `Q` to have interface `QUEUE` but does not specify an implementation. The interface supplied for unit `Q` is sufficient to compile unit `C`: The top-level declaration in unit `C` is compiled in a context binding a single structure `Queue`.

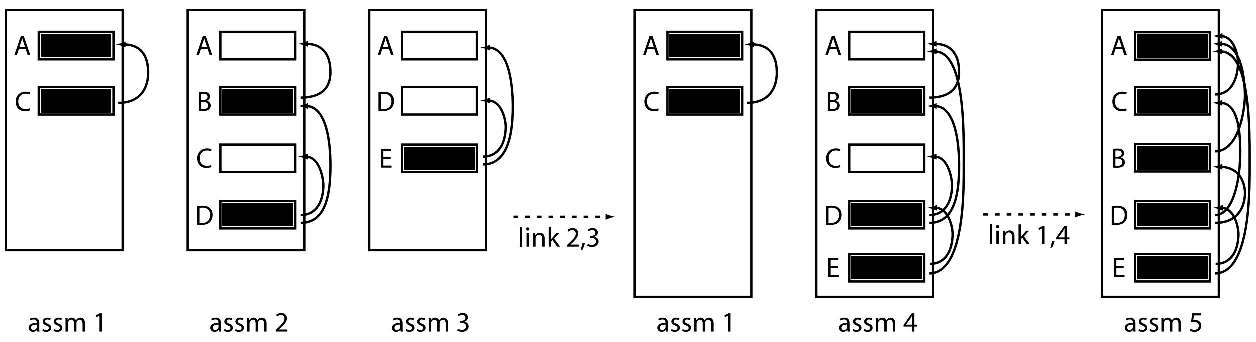

Figure 2: An example program being linked. The letters are unit names. Filled boxes correspond to unit implementations and lines from a filled box up to other boxes indicate opened units. In the first step, three assemblies are separately developed and the constituent units separately compiled. We can partially link assemblies 2 and 3 to give us a fourth assembly. This assembly still has unimplemented units, so it cannot be made into an executable yet. Linking it with assembly 1, however, resolves all of these dependencies and so an executable can be produced.

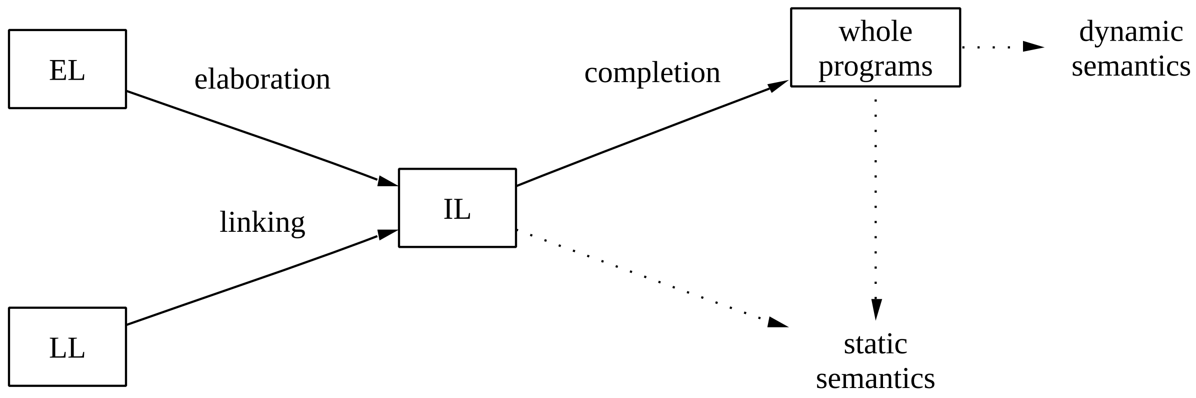

Figure 3: Organization of the technical material in this proposal. Link scripts and source units, interfaces, and assemblies are given meaning via translation into the internal language. The relations and the IL employ stubs that are realized for TD and TS.

to the underlying semantics. These stubs are separately *realized* in two forms, one corresponding to *The Definition of Standard ML* [MTHM97], which we will abbreviate by TD, the other corresponding to the Typed Semantics of Standard ML [HS00], which we will abbreviate by TS. This organization permits us to provide an interpretation of separate compilation in terms of either well-known semantic framework, and also suggests an implementation strategy that is compatible with all known compiler architectures for Standard ML. The semantics specifies when one unit depends on another, when an assembly is complete, and hence may be used to build an executable, and the order of side effects. This provides a clear criterion for the correctness of an implementation, and for the compatibility of different implementations.

The semantics of separate compilation is given in terms of three languages and various relations among them. The *internal language* (IL), is a language of "compiled" units, interfaces and assemblies. IL stubs provide the syntax and static semantics for elementary compiled units and interfaces, based on the underlying semantic framework. The *external language* (EL) is the source language of units, interfaces and assemblies. Its syntax builds on SML and its meaning is specified by an *elaboration* translation into the IL. Elaboration stubs translate elementary SML source code to compiled units and interfaces, again based on the underlying semantics. The *linking language* (LL) builds on the IL and is given meaning via a *linking* translation into the IL. Linking stubs specify when an elementary compiled unit or interface makes reference to another unit. A *completion* stub translates a fully linked, compiled assembly to a *program*, which has a dynamic semantics specifying its execution. Figure 3 summarizes the situation.

This organization avoids commitment to specific interpretations of "elaboration" or "completion" so as to ensure compatibility with various semantic and implementation strategies. For example, a whole-program compiler might define elaboration to perform only type checking, deferring code generation to the completion phase. Alternatively, standard separate compilation may be performed by specifying elaboration to include code generation, and completion to include only resolution of external references.

1.2 Rationale

Several major design principles informed the development of this proposal:

A language, not a tool. We propose an extension to the Standard ML language to support separate compilation, rather than a tool to implement it. The extension is defined by a semantics that extends the semantics of SML to provide a declarative description of the meanings of the language constructs. The semantics provides a clear correctness criterion for implementations to ensure source-level compatibility among them.

Flexibility. A compilation unit consists of any sequence of top-level bindings, including signature and functor declarations.[2] However, since Standard ML lacks syntactically expressible signatures, some units cannot be separately compiled from one another, and must therefore be considered together in a single assembly.

Simplicity. The design provides only the minimum functionality of a separate compilation system. It omits any form of compilation parameters, conditional compilation directives, or compiler directives. We leave for future work the specification of such machinery.[3]

Conservativity. The semantics of Standard ML should not be changed by the introduction of separate compilation. In particular, we do not permit "circular dependencies" or similar concepts that are not otherwise expressible in the language. This ensures that existing compilers should not be disturbed by the proposed extension beyond what is required to implement the extension itself.

Explicit dependencies. The dependencies among units and assemblies is explicitly specified, not inferred. The chief reason for this is that dependencies among units may not be syntactically evident—for example, the side effects of one unit may influence the behavior of another. Moreover, there are, in general, many ways to order effects consistently with observed dependencies, and these orderings need not be equivalent. A lesser reason is that supporting dependency inference requires restrictions on compilation units that are not semantically necessary, reducing flexibility.

No added sharing. Unit references are definite; unit names have global scope and cannot be shadowed. This ensures that the use of separate compilation does not induce the need for any additional sharing specifications.

Environment independence. The separate compilation system is defined independently of any environment in which it might be implemented. The design speaks in terms of linguistic and semantic entities, rather than implementation-specific concepts such as files or directories.

The remainder of this proposal is organized as follows. In Section 2 we describe the extension's implementation in the TILT compiler, presenting a concrete syntax and command-line interface for separate compilation. We discuss the implementation first in order that the development of the formalism that follows can be grounded in concrete intuitions. In Section 3 we give the syntax and semantics of the extension, in a form independent of the underlying semantic framework. In

[2] Consequently, units cannot be identified with structures in the sense of the Standard ML module system.

[3] The TILT compiler includes such facilities, and might serve as a basis for a future extension of the proposal.

unit	A sequence of SML top-level declarations with free identifiers resolved by reference to a list of opened units.
interface	The type of a unit: A sequence of top-level specifications with free identifiers resolved by reference to a list of opened units.
assembly	An independently meaningful sequence of unit and interface declarations. An assembly must specify an interface for any externally defined unit to which it refers.
link script	Description of how to link a sequence of assemblies to form another.
external language	The language of source assemblies, units, and interfaces.
linking language	The language of link scripts.
internal language	The language of compiled assemblies, units, and interfaces. It serves as the target language of elaboration and linking.
elaboration	Type-checking and transformation from external to internal form.
linking	Creation of an assembly from a sequence of assemblies.
completion	Creation of an executable from an assembly with no external references.

Figure 4: Glossary of main concepts

Section 4 we realize the semantics for TS, and in Section 5 we do the same for TD. In Section 6 we review related work.

For handy reference, a glossary of the main concepts used in this proposal is given in Figure 4.

2 Implementation in TILT

In this section, we discuss the separate compilation language implemented by the TILT compiler for Standard ML [TIL]. Except for minor differences and extensions, TILT implements separate compilation as described by the TS realization of the semantics (Sections 3 and 4).

Most of this proposal concerns the abstract syntax and semantics of separate compilation. A concrete syntax is necessary, too, but we leave a rigorous treatment to future work. For the sake of discussion, we give in Figures 5 and 6 a concrete syntax based on that used in TILT. Optional elements are enclosed in single angle brackets. The nonterminals *filename*, *msg*, and *test* correspond to a small language of strings, integers, and booleans. Expressions in this language can access compiler parameters and environment variables. (Assembly and interface files are lexically similar to SML.)

Assembly Files. A concrete assembly, or assembly file, declares a list of units and interfaces. Top-level declarations—SML source code—and specificiations must be in their own files. The

assembly	::=		empty
		assembly assmdec	
assmdec	::=	`interface` *intid* = *intexp*	interface definition
		`unit` *unitid* : *intexp*	unit description
		`unit` *unitid* ⟨: *intexp*⟩ = *source*	unit definition
		`include` *filename*	
		`#if` *test assembly* ⟨*cc*⟩ `#endif`	conditional
		`#error` *msg*	abort
intexp	::=	*intid*	
		source	
source	::=	*filename* ⟨`{` *unitids* `}`⟩	
unitids	::=		empty
		unitids unitid	
cc	::=	`#else` *assembly*	
		`#elif` *test assembly* ⟨*cc*⟩	

Figure 5: Concrete syntax of assembly files

unitfile	::=	*topdec*	top-level declaration
interfacefile	:=	*topspec*	top-level specification
topspec	::=	*spec*	basic
		`functor` *funspec*	functor
		`signature` *sigbind*	signature
		`infix` ⟨*d*⟩ *vids*	fixity
		`infixr` ⟨*d*⟩ *vids*	
		`nonfix` *vids*	
		$topspec_1$ ⟨;⟩ $topspec_2$	sequence
funspec	::=	*funid*(*strid* : *sigexp*) : $sigexp'$	
		funid(*spec*) : *sigexp*	
		funspec `and` *funspec*	
vids	::=	*vid* ⟨*vids*⟩	

Figure 6: Concrete syntax of unit and interface files

contents of a named file and a list of opened units written in curly braces constitute a concrete unit or interface. The opened units may be omitted as a short-hand for opening every unit declared to that point in the assembly file, in the order they appear. To open no units, an explicit {} is required.

TILT permits an assembly to be split into one or more assembly files and supports conditional compilation at the level of unit and interface declarations. Assembly files may include other assembly files and the programmer may specify a list of assembly files on the command-line. TILT avoids including the same file more than once by syntactically interpreting relative paths and comparing the resulting file names. This is used to detect "include cycles" and to permit two included assembly files to include a third.

The assembly file parser in TILT produces an assembly in the (much simpler) EL abstract syntax.[4] In translating from concrete to abstract syntax, the parser eliminates conditional compilation, incorporates included assembly files, and so on. A concrete assembly is, of course, an assembly. It must be independently meaningful, specifying an interface for any externally defined unit to which it refers, and may have at most one declaration for each unit or interface identifier. Thus, the parser must combine concrete assemblies similar to how the linker of Section 3.4 combines compiled assemblies.[5] The chief difference is that the parser can not check interface equivalence, which can only be judged after elaboration. It considers two concrete interfaces equivalent if they are identical (same file contents and lists of opened units). This is a conservative approximation of semantic interface equivalence. A reasonable alternative would be for the parser to residuate a list of interface equivalence constraints that must be checked during compilation.

TILT can not generate SML source files using tools like **ml-yacc** and **ml-lex** or shell recipes. Such support could be added to the assembly file parser with little difficulty.

Fixity. TILT interface files may contain fixity declarations. In this proposal, we do not formalize parsing SML concrete syntax to abstract syntax, so we do not give a semantics to fixity declarations. However, we note that our intention is to permit a program to be split into units between any two top-level declarations and for interfaces ascribed to those units to mediate interactions among them. This essentially forces the following treatment of fixity declarations.

Concrete interfaces may include fixity declarations so that they can describe concrete units. IL interfaces must include fixity information so that interface ascription (defined in terms of IL interfaces) can check that a unit provides at least the fixity information in its ascribed interface. Fixity declarations influence IL interface equivalence and sub-interface relations. Finally, the fixity information in any interface must be activated when opening a unit so that parsing of its dependents is performed in a manner consistent with integrated compilation.

Command-Line. Link scripts are implicit in the TILT command-line. There are three ways to invoke TILT:

- `tilt assembly` parses the assembly file `assembly` and compiles the resulting EL assembly to an IL assembly.
- `tilt -o exe assembly` compiles `assembly` and completes the result to an executable `exe`.

[4] Please see Section 3.1 for a discussion of the abstract syntax.

[5] A common scenario is for assembly file L_1 to implement the units in a library, for assembly file L_2 to implement a second library and to describe those units from L_1 needed for its implementation, and for assembly file A to include both L_1 and L_2. Parsing A must produce an assembly that declares the units in L_1 once.

- `tilt -l lib assembly` compiles `assembly` and puts the resulting IL assembly in a directory `lib` along with assembly files that describe it.

By default, TILT does not use selective linking. This can be changed with a command-line option. For example, the command

```
tilt -c Main -o exe assembly
```

specifies that `exe` should contain only unit `Main` and any units that it needs.

Having TILT copy an IL assembly into a directory `lib` is entirely optional. The benefit of doing so is that TILT writes assembly files to provide two views of the units in `lib`. The first declares all of the units with their implementations. The second declares all of the units but does not specify any implementations. A third assembly file uses the conditional compilation mechanism to include the first or the second depending on whether or not the compiler is completing an executable. By convention, an assembly that needs the units in `lib` includes this third file. From the point of view of the including assembly, `lib` consists of an up-to-date collection of separately compiled units. When the including assembly is completed, the implementations of these units is obtained from `lib`. When the including assembly is copied to its own `libdir`, the copy contains descriptions of the units in `lib` but not their implementations. (TILT uses this mechanism to make its implementation of the Standard Basis Library available to every assembly file.)

Standard Basis Library. The Standard Basis library is automatically included as part of every assembly file. Moreover, each interface and unit implicitly opens those units that provide the standard top-level environment. All structures and functors in the Standard Basis are defined in units of the same name as the structure or functor. Most signatures defined in the Standard Basis are defined in units of the same name as the signature, with the exception of the signatures `IO`, `OS`, and `SML90`, which reside in units named `IO_SIG`, `OS_SIG`, and `SML90_SIG`, respectively.

Compilation. TILT supports parallel compilation, where several machines work together to compile the interfaces and units in a single assembly. A unit or interface is ready for compilation as soon as the IL interfaces of its opened units are up-to-date. Since interface ascription is coercive, the dependents of a unit with an ascribed interface do not have to wait for the unit to be compiled. Less important, the dependents of a large unit with an inferred interface can be compiled once the unit is elaborated (and its IL interface is written to disk). They do not have to wait for the unit to be fully compiled to an object file. The semantics of separate compilation should enable us to state and check the correctness of parallel compilation as well as, with a little more work, the use of cut-off incremental recompilation [ATW94] in TILT.

Examples. We give examples of the concrete syntax in Figures 7 and 8. (The assembly file `echo.assm` makes use of declarations in the implicitly included Basis Library.)

3 Syntax and Semantics

In this section we define the internal, external, and linking languages used to give the semantics of separate compilation. The meta-theory of the semantics and its realization to TD and TS is relegated to Appendix F.

(echo.sml *)*

```
fun echo (ss:string list) : unit =
    (case ss of
        nil => ()
    |   s::nil => print s
    |   s::ss => (print s; print " "; echo ss))

val () =
    (case (CommandLine.arguments()) of
        "-n" :: args => echo args
    |   args => (echo args; print "\n"))

val () = OS.Process.exit OS.Process.success
```

(echo.assm *)*

```
unit Echo = "echo.sml" { CommandLine OS }
```

Figure 7: An implementation of the Unix `echo` command. The command `tilt -o echo.exe -c Echo echo.assm` creates an executable.

```
(* queue-sig.sml *)
signature QUEUE =
sig
 type 'a queue
 val empty : 'a queue
 val push : 'a * 'a queue -> 'a queue
end
```

```
(* queue.sml *)
structure Queue :> QUEUE =
struct
 type 'a queue = 'a list
 val empty = nil
 val push = op ::
end
```

```
(* queue.int *)
structure Queue : QUEUE
```

```
(* main.sml *)
val q = Queue.empty
val q' = Queue.push (0, q)
```

```
(* lib.assm *)
interface QSIG = "queue-sig.sml"
unit QSIG : QSIG = "queue-sig.sml"
interface QUEUE = "queue.int" { QSIG }
unit Q : QUEUE = "queue.sml" { QSIG }
```

```
(* client.assm *)
interface QSIG = "queue-sig.sml"
unit QSIG : QSIG
interface QUEUE = "queue.int" { QSIG }
unit Q : QUEUE
unit C = "main.sml" { Q }
```

Figure 8: Simple assemblies. The command `tilt client.assm` compiles the client, `tilt lib.assm` compiles the library separately from the client, and `tilt -o queue.exe lib.assm client.assm` links the compiled assemblies together and completes them to an executable. (The order of assemblies on the command-line corresponds to the order of IL assemblies in the implicit link script. The link would fail if the client preceded the library.)

$$
\begin{array}{rcll}
\mathit{assembly} & ::= & \cdot & \\
 & & \mathit{assembly}, \mathit{intid} = \mathit{intexp} & \text{interface definition} \\
 & & \mathit{assembly}, \mathit{unitid} : \mathit{intexp} & \text{unit description} \\
 & & \mathit{assembly}, \mathit{unitid}\ \langle : \mathit{intexp} \rangle = \mathit{unitexp} & \text{unit definition} \\
\mathit{unitexp} & ::= & \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} & \\
\mathit{intexp} & ::= & \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topspec} & \\
 & & \mathit{intid} & \\
\mathit{topspec} & ::= & \mathit{spec} & \text{basic} \\
 & & \mathsf{functor}\ \mathit{funspec} & \text{functor} \\
 & & \mathsf{signature}\ \mathit{sigbind} & \text{signature} \\
 & & \mathit{topspec}_1\ \mathit{topspec}_2 & \\
\mathit{funspec} & ::= & \mathit{funid}(\mathit{strid} : \mathit{sigexp}) : \mathit{sigexp}'\ \langle \mathsf{and}\ \mathit{funspec} \rangle & \\
\mathit{unitids} & ::= & \mathit{unitid}_1 \cdots \mathit{unitid}_n &
\end{array}
$$

Figure 9: Abstract syntax of the external language

3.1 External Language

The abstract syntax of the EL is given in Figure 9. The syntactic categories *topdec*, *spec*, *sigbind*, *funid*, *strid*, and *sigexp* are inherited from TDEL.[6] The syntactic categories *unitid* (unit identifiers) and *intid* (interface identifiers) are presumed to be disjoint from each other and from all other identifier classes. We require that no *topspec* or *funspec* may specify the same identifier twice. (We give meaning to the EL through elaboration to the IL in Section 3.3.)

Two possibly surprising aspects of the EL are that units and interfaces do not stand alone but are declared in assemblies and that within assemblies, unit and interface identifiers do not obey the usual rules of lexical scoping.

Units and interfaces have no meaning independent of an assembly: They contain free identifiers and can not be compiled in isolation. A unit or interface may make reference to another unit, opening it by name and obtaining its interface from the ambient assembly. In addition, the interface for a unit may make reference to an abstract type defined in another unit. To do away with assemblies, it seems necessary for each unit or interface to describe its entire compilation context, comprising an interface for each opened unit and, transitively, for any units whose abstract types are referenced. This approach would place a tremendous annotation burden on the programmer.

To properly resolve external references, the linker must know when two occurrences of a unit name refer to the same unit. In the proposed extension, unit names have global scope and cannot be shadowed so every reference to a unit named *unitid* refers to the *same* unit. (For consistency, EL interface names cannot be shadowed.) If unit names could be shadowed, or if two assemblies using the same name to refer to different units could be linked together, then unit references would be indefinite: A reference to a unit named *unitid* would refer to *some* unit with that name. The linker would need help from the programmer in matching external references to unit implementations.

3.2 Internal Language

The IL syntax is given in Figure 10. The syntactic categories *assm* and *adecs* specify lists of elements. We adopt the following notation for these and other lists:

[6]The external languages of *The Definition* and *The Typed Semantics* differ. We refer to them as TDEL and TSEL when it is necessary to distinguish between them.

$$
\begin{array}{rcll}
\mathit{assm} & ::= & \cdot & \\
& & \mathit{assm}, \mathit{unitid} : \mathit{intf} & \text{unit description} \\
& & \mathit{assm}, \mathit{unitid} : \mathit{intf} = \mathit{unite} & \text{unit definition} \\
\mathit{unite} & ::= & \langle\mathsf{internal}\rangle\ \mathsf{require}\ \mathit{unitids}\ \mathsf{in}\ \mathit{impl} & \\
\mathit{adecs} & ::= & \cdot & \\
& & \mathit{adecs}, \mathit{adec} & \\
\mathit{adec} & ::= & \mathit{unitid} : \mathit{intf} & \text{unit description}
\end{array}
$$

Figure 10: Abstract syntax of the internal language

intf	compiled interface
impl	compiled unit
Γ	context
$\Gamma \vdash \mathit{intf} : \mathsf{Intf}$	*intf* is well-formed
$\Gamma \vdash \mathit{impl} : \mathit{intf}$	*impl* has interface *intf*
$\Gamma \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}$	interface equivalence
$\Gamma \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}$	*intf* is a sub-interface of intf'
$\mathit{adecs} \vdash \Gamma$	Γ declares units in *adecs*
$\vdash \Gamma\ \mathsf{ok}$	Γ is well-formed

Figure 11: Internal language stubs

- We denote by $(\cdot, \cdot)$ the operation of syntactic concatenation; for example, $\mathit{assm}, \mathit{assm}'$.
- We sometimes use pattern matching at the left end of the list, writing $\mathit{adec}, \mathit{adecs}$ to match the first binding in the list.
- We usually omit the initial $\cdot$; for example, $\mathit{adec}_1, \ldots, \mathit{adec}_n$.

In order to support the two different semantic frameworks for SML, a few IL syntactic categories and judgements are stubs. These appear in Figure 11. For the syntax, the relevant stubs are the syntactic categories for compiled units and interfaces, *impl* and *intf*.

For example, the assembly given in Figure 1 elaborates to:

$$
\begin{array}{l}
\mathsf{basis} : \mathit{intf}_{\mathit{basis}}, \\
\mathtt{Q} : \mathit{intf}_1, \\
\mathtt{C} : \mathit{intf}_2 = \mathsf{internal\ require}\ \mathtt{Q}\ \mathsf{in}\ \mathit{impl}_2,
\end{array}
$$

where intf_1 is the compiled interface `QUEUE`, intf_2 is the compiled interface inferred for unit `C`, and impl_2 is the compiled unit `C`. (The basis unit is discussed in Section 3.3.)

Beyond the fact that source code is replaced by compiled code, the main differences between the EL and the IL are as follows. First, the IL does not support named interfaces; instead, every unit declaration comes explicitly with its interface. Second, units may be marked internal and the linker will prevent them from being used to satisfy external dependencies. The elaborator marks units with inferred interfaces internal. Third, instead of the EL open mechanism, the IL has require. Selective linking respects the *initialization dependencies* of units and the and *reference dependencies* of units and interfaces [HP05]. The require clause for a unit records those units that must be retained for their effects whenever the unit is retained (initialization dependencies). The EL does not distinguish these two dependency relations, using open for both, so the appearance of `Q` in the require clause for `C` simply records the fact that `Q` is opened in the source.

Judgement...	*Meaning...*
$\mathit{adecs} \vdash \mathit{assm}\ \mathsf{ok}$	*assm* is well-formed
$\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$	*intf* is well-formed
$\mathit{adecs} \vdash \mathit{unite} : \mathit{intf}$	*unite* has interface *intf*
$\mathit{adecs} \vdash \mathit{impl} : \mathit{intf}$	*impl* has interface *intf*
$\mathit{adecs} \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}$	interface equivalence
$\mathit{adecs} \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}$	*intf* is a sub-interface of intf'
$\vdash \mathit{adecs}\ \mathsf{ok}$	*adecs* is well-formed

Figure 12: Internal language judgements

The judgement forms for the static semantics of the IL are given in Figure 12. Type-checking takes place relative to an IL context, *adecs*, that records declared units. A context is well-formed if no *unitid* is declared more than once and every *intf* is well-formed. An assembly, *assm* is well-formed if, in addition, no *unitid* is used before it is declared and every *impl* is well-formed. (The rules for the static semantics are given in Appendix A.)

3.3 Elaboration

Elaboration type-checks a source assembly, unit, or interface and, if it is well-typed, translates it to compiled form. Elaboration is defined relative to the underlying semantic framework using the stubs summarized in Figure 13. First, we need an interface for the top-level basis unit that can be assumed by every Standard ML program. This unit defines the built-in types of the language, and the built-in exceptions, such as Match, that are required by the underlying framework. The elaborator ensures this unit is implicitly described in every EL assembly and opened for use in every EL unit and interface. (It is implemented by the underlying semantics prior to evaluation.) Second, we need a way to elaborate the source code of a unit (a *unitexp*) in a specified context, generating a compiled unit and interface for it. Similarly, we need to be able to compile an EL interface to an IL interface in a specified context. Third, to support coercive interface ascription, we require an ascription operation that checks a compiled unit against a compiled interface and generates a new unit satisfying that interface.

Elaboration takes place relative to an elaboration context, *edecs*, that records the result of elaborating the preceding unit and interface declarations. The syntax of elaboration contexts is defined in Figure 14, and the elaboration judgement forms are given in Figure 15. For the most part, elaboration is a straightforward process making use of the stubs to elaborate units and interfaces. (The rules for elaboration are given in Appendix B.)

$\mathit{intf}_{\mathit{basis}}$	basis interface
$\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \leadsto \mathit{impl} : \mathit{intf}$	unit elaboration
$\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topspec} \leadsto \mathit{intf}$	interface elaboration
$\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \leadsto \mathit{impl}$	interface ascription

Figure 13: Elaborator stubs

edecs	$::=$	$\cdot$	
		$\mathit{edecs}, \mathit{adec}$	unit description
		$\mathit{edecs}, \mathit{intid} : \mathsf{Intf} = \mathit{intf}$	interface definition

Figure 14: Elaboration contexts

Judgement...	*Meaning...*
$\vdash \mathit{assembly} \rightsquigarrow \mathit{assm}; \mathit{edecs}$	assembly elaboration
$\mathit{edecs} \vdash \mathit{unitexp} \rightsquigarrow \mathit{unite} : \mathit{intf}$	unit elaboration
$\mathit{edecs} \vdash \mathit{intexp} \rightsquigarrow \mathit{intf} : \mathsf{Intf}$	interface elaboration
$\mathit{edecs} \vdash \mathit{unite}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{unite}$	interface ascription
$\mathit{edecs} \vdash \mathit{adecs}$	*adecs* declares units in *edecs*
$\vdash \mathit{edecs}\ \mathsf{ok}$	*edecs* is well-formed

Figure 15: Elaboration judgements

3.4 Linking and Completion

The syntax of the linking language is given in Figure 16. A link script consists of a series of assemblies to link together and an optional selective linking directive. Selective linking retains only those units in the linked assembly that are required by a list of target units, respecting initialization and reference dependencies.

The linker stubs are described in Figure 17. We require a class of executable programs *prog*, and a judgement for their well-formedness. We need to query a unit or interface to see if a unit identifier is free in it (reference dependencies). Finally, we need a way to convert an assembly with no external dependencies (except for the basis unit) to an executable *prog*.

Linking is a two-step process. The first step combines the assemblies in a link script to a single, well-formed assembly that declares all of their units. Combination takes place relative to a combination context, *cdecs*, that records declared units and whether or not they are internal. The second step selects those units in the combined assembly that are required by the link script (discarding the rest). Selection takes place relative to a fixed dependency context, *deps*, comprising the combined assembly and a list of targets. We give the syntax of combination and dependency contexts in Figure 18 and the judgement forms for linking and completion in Figure 19. The rules for linking presuppose that the link script is well-formed. The rules for combination examine each unit declaration in the link script from left-to-right. The first declaration for a unit is kept whereas subsequent declarations are checked but discarded. The rules for selection examine each unit declaration in the combined assembly from left-to-right, discarding those that are not required. A unit is required if it is a target of the link script; the code/interface of a required unit makes reference to it; or it is listed in the $\mathsf{require}$ clause of a required unit. (The rules for linking are given in Appendix C.)

4 TS Realization

After a brief review of *The Typed Semantics of Standard ML* [HS00, HS97], we realize the semantics of separate compilation for TS (Sections 4.1–4.3) and apply the dynamic semantics of the TS internal language to programs arising from complete assemblies (Section 4.4).

lscript	::=	combine *assms*	link
	::=	from *assms* select *unitids*	link selectively
assms	::=	$\cdot$	
		assm; *assms*	

Figure 16: Abstract syntax of the linking language

prog	executable programs
$\vdash$ *prog* ok	*prog* is well-formed
$\vdash$ *intf* requires *unitid*	*unitid* is free in *intf*
$\vdash$ *impl* requires *unitid*	*unitid* is free in *impl*
$\vdash$ *assm* $\rightsquigarrow$ *prog*	completion

Figure 17: Linker stubs

cdecs	::=	$\cdot$	
		$\mathit{cdecs}, \mathit{unitid} :_{\langle i \rangle} \mathit{intf}$	unit description
deps	::=	*assm*; *unitids*	combined assembly and targets

Figure 18: Combination and dependency contexts

Judgement...	*Meaning...*
$\vdash$ *lscript* $\rightsquigarrow$ *assm*	linking
cdecs $\vdash$ *assms* $\rightsquigarrow$ *assm*	combination
$\mathit{adecs} \vdash_{\mathit{deps}} \mathit{assm} \rightsquigarrow \mathit{assm}'$	selection
$\vdash$ *deps* requires *unitid*	required units
$\vdash$ *assm* $\rightsquigarrow$ *prog*	completion (stub)
adecs $\vdash$ *assm* complete	*assm* is complete
cdecs $\vdash$ *adecs*	*adecs* declares units in *cdecs*
$\vdash$ *lscript* ok	*lscript* is well-formed
adecs $\vdash$ *assms* ok	*assms* is well-formed
$\vdash$ *cdecs* ok	*cdecs* is well-formed
$\vdash$ *deps* ok	*deps* is well-formed

Figure 19: Linking judgements

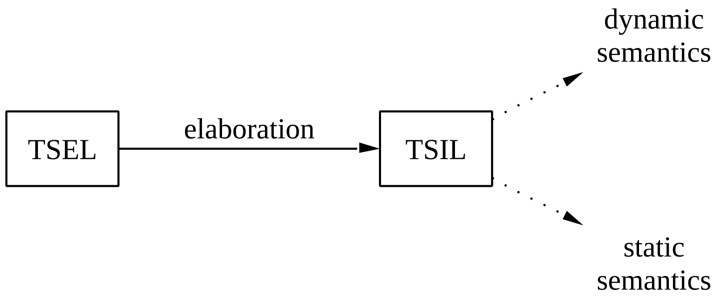

Figure 20: Organization of *The Typed Semantics of Standard ML*

TS defines TSEL through elaboration into an explicitly typed λ-calculus called the TS internal language (TSIL); the situation is summarized in Figure 20. The TSIL has a coherent static and dynamic semantics, is rich enough to keep the translation simple, and is small enough to be tractable. The TSIL is divided into a *core* level of expressions, constructors, and kinds and a *module* level of modules and signatures. Both the TSIL and the translation benefit from an emphasis on a few primitive notions. As one example, the "type generativity" of Standard ML and such core-level constructs as polymorphism, datatypes, and equality types are encoded as uses of the TSIL module system. These encodings are quite natural so that while they serve to simplify the TSIL, they do not unduly complicate elaboration. Another example is the distinction between labels (that correspond to Standard ML identifiers) and variables (that may be alpha-varied). This distinction admits a treatment of the scoping rules of Standard ML, including types that *apparently* escape their scope, as in the local datatype example in the introduction.

The judgement forms of the TSIL static semantics are given in Figure 21 and the syntax of the TSIL is summarized in Figure 22. (The syntax of values—exp_v, $sbnds_v$, and mod_v—is omitted, but note that *path*s—variables and projections from module variables—are values.) At the core level, constructors classify expressions and kinds classify constructors; at the module level, signatures classify modules. There are a number of points of interest in the sequel. First, TSIL signature equivalence is not coercive; for example, if $decs \vdash sdecs \equiv sdecs'$, then $sdecs$ and $sdecs'$ declare the same components, in the same order, with the same labels, and corresponding type components are equivalent. Second, the judgement $decs \vdash mod : sig$ can be used to obtain the "selfified" signature of a bound structure variable. For example, if var is bound to a structure with one abstract type component, $\mathtt{t}$, then the judgement

$$decs_1, var : [\mathtt{t} \rhd var_t : \Omega], decs_2 \vdash var : sig$$

holds where the $\mathtt{t}$ component of $sig = [\mathtt{t} \rhd var_t : \Omega = var.var_t]$ is equivalent to the bound opaque type. Finally, the TSIL static semantics (and the TS elaborator) is non-deterministic. As one example, the preceding judgement also holds where the $\mathtt{t}$ component of $sig = [\mathtt{t} \rhd var_t : \Omega]$ is kept abstract and is *not* equivalent to the bound type.

The TSIL dynamic semantics is a small-step, call-by-value operational semantics presented as a rewriting system on states of an abstract machine [HS00, pages 350–352]. A *state* has the form $\Sigma = (\Delta, \sigma, E)$, where Δ is a typing context ($decs$) for locations and tags, σ is a finite mapping from locations typed in Δ to values, and E is an *evaluation context* comprising an expression or module with a single hole that is replaced by the phrase being evaluated. The dynamic semantics is a transition relation $\Sigma \hookrightarrow \Sigma'$ between states.

Judgement...	*Meaning...*
$\vdash \mathit{decs}\ \mathsf{ok}$	decs is well-formed
$\mathit{decs} \vdash \mathit{dec}\ \mathsf{ok}$	dec is well-formed
$\mathit{decs} \vdash \mathit{bnd} : \mathit{dec}$	bnd has declaration dec
$\mathit{decs} \vdash \mathit{knd} : \mathsf{Kind}$	knd is well-formed
$\mathit{decs} \vdash \mathit{con} : \mathit{knd}$	con has kind knd
$\mathit{decs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}$	constructor equivalence at kind knd
$\mathit{decs} \vdash \mathit{exp} : \mathit{con}$	exp has type con
$\mathit{decs} \vdash \mathit{sdecs}\ \mathsf{ok}$	sdecs is well-formed
$\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig}$	sig is well-formed
$\mathit{decs} \vdash \mathit{sdecs} \leq \mathit{sdecs}'$	component-wise subtyping
$\mathit{decs} \vdash \mathit{sig} \leq \mathit{sig}' : \mathsf{Sig}$	signature subtyping
$\mathit{decs} \vdash \mathit{sdecs} \equiv \mathit{sdecs}'$	component-wise equivalence
$\mathit{decs} \vdash \mathit{sig} \equiv \mathit{sig}' : \mathsf{Sig}$	signature equivalence
$\mathit{decs} \vdash \mathit{sbnds} : \mathit{sdecs}$	sbnds has declaration list sdecs
$\mathit{decs} \vdash \mathit{mod} : \mathit{sig}$	mod has signature sig
$\mathit{decs} \vdash \mathit{exp} \downarrow \mathit{con}$	exp is valuable with type con
$\mathit{decs} \vdash \mathit{mod} \downarrow \mathit{sig}$	mod is valuable with signature sig

Figure 21: TSIL judgements

knd	$::=$	$\cdots$	kinds
		Ω	kind of types
con	$::=$	$\cdots$	constructors
		$\{\mathit{lab}_1 : \mathit{con}_1, \ldots\}$	record type
		Tagged	extensible sum type
		$\mathit{con}\ \mathsf{Tag}$	exception-tag type
		$\mathit{mod}_v.\mathit{lab}$	module projection
exp	$::=$	$\cdots$	expressions
		$\{\mathit{lab}_1 = \mathit{exp}_1, \ldots\}$	record expression
		$\mathsf{raise}^{\mathit{con}}\ \mathit{exp}$	raise expression
		$\mathsf{new\_tag}[\mathit{con}]$	extend type Tagged
		$\mathsf{tag}(\mathit{exp}, \mathit{exp})$	injection into Tagged
		$\mathit{mod}.\mathit{lab}$	module projection
mod	$::=$	var	module variables
		$[\mathit{sbnds}]$	structure
		$\lambda \mathit{var} : \mathit{sig}.\mathit{mod}$	functor
		$\mathit{mod}\ \mathit{mod}'$	functor application
		$\mathit{mod}.\mathit{lab}$	structure projection
		$\mathit{mod} : \mathit{sig}$	signature ascription
sbnds	$::=$	$\cdot$	structure field bindings
		$\mathit{sbnds}, \mathit{sbnds}$	
sbnd	$::=$	$\mathit{lab} \triangleright \mathit{bnd}$	
bnd	$::=$	$\mathit{var} = \mathit{con}$	constructor binding
		$\mathit{var} = \mathit{exp}$	expression binding
		$\mathit{var} = \mathit{mod}$	module binding
sig	$::=$	$[\mathit{sdecs}]$	structure signature
		$(\mathit{var} : \mathit{sig}) \rightharpoonup \mathit{sig}'$	partial functor signature
		$(\mathit{var} : \mathit{sig}) \rightarrow \mathit{sig}'$	total functor signature
sdecs	$::=$	$\cdot$	structure field declarations
		$\mathit{sdecs}, \mathit{sdec}$	
sdec	$::=$	$\mathit{lab} \triangleright \mathit{dec}$	
decs	$::=$	$\cdot$	declaration lists
		$\mathit{decs}, \mathit{dec}$	
dec	$::=$	$\mathit{var} : \mathit{con}$	expression variable dec.
		$\mathit{var} : \mathit{sig}$	module variable dec.
		$\mathit{var} : \mathit{knd}$	opaque type dec.
		$\mathit{var} : \mathit{knd} = \mathit{con}$	transparent type dec.
		$\mathit{loc} : \mathit{con}$	typed locations
		$\mathsf{tag} : \mathit{con}$	typed exception tags
phrase	$::=$	$\mathit{exp} \mid \mathit{mod} \mid \mathit{con}$	phrases
class	$::=$	$\mathit{con} \mid \mathit{sig} \mid \mathit{knd}$	phrase classifiers

Figure 22: TSIL syntax (summary)

Judgement...	*Meaning...*
$sdecs \vdash strdec \leadsto sbnds : sdecs'$	declaration elaboration
$sdecs \vdash strexp \leadsto mod : sig$	structure expression elaboration
$sdecs \vdash spec \leadsto sdecs'$	signature specification elaboration
$sdecs \vdash sigexp \leadsto sig : \mathsf{Sig}$	signature expression elaboration
$sdecs \vdash_{\mathsf{ctx}} labs \leadsto path : class$	lookup in *sdecs*
$sdecs \vdash_{\mathsf{ctx}} labs \leadsto path$	
$decs; path{:}sig \vdash_{\mathsf{sig}} labs \leadsto labs' : class$	lookup in signature
$sig \vdash_{\mathsf{sig}} lab \leadsto labs'$	
$decs \vdash_{\mathsf{inst}} \leadsto [sbnds_v] : [sdecs']$	polymorphic instantiation
$decs \vdash_{\mathsf{sub}} path : sig_0 \preceq sig \leadsto mod : sig'$	coercion compilation
$decs; path{:}sig_0 \vdash_{\mathsf{sub}} sdec \leadsto sbnd : sdec'$	
$sig \vdash_{\mathsf{wt}} labs := con : knd \leadsto sig' : \mathsf{Sig}$	impose definition
$sig \vdash_{\mathsf{sh}} labs := labs' : knd \leadsto sig' : \mathsf{Sig}$	impose sharing

Figure 23: TS elaboration judgements (summary)

The judgement forms of the TS elaborator are summarized in Figure 23. (The judgements for elaborating core constructs are omitted.) The elaboration judgements perform type checking, type reconstruction, and translation to the TSIL. There is no elaboration judgement for TDEL top-level declarations because TSEL does not include them. Instead, TSEL permits functor declarations within structure declarations and TS treats signature declarations as abbreviations that are expanded prior to TS elaboration. We resolve these differences in Section 4.2. The identifier lookup judgements address the scoping rules of Standard ML. To handle "open" structures, the lookup rules descend into modules with starred labels ($lab^\star$). The other judgements in Figure 23 perform polymorphic instantiation, supply explicit coercions to account for Standard ML signature matching, and replace Standard ML `sharing` specs and `where type` signature patching with IL transparent type declarations.

4.1 Realization of the Internal Language for TS

We realize the IL syntactic stubs for TS in Figure 24. A compiled unit is a TSIL module that binds the top-level type, expression, structure, and functor components of the unit. Signature definitions do not appear in compiled units.

A compiled interface has the form $var : [sdecs]; tdecs$. The structure signature $[sdecs]$ describes compiled units with this interface and the top-level declarations list *tdecs* contains their signature definitions. The bound variable *var* has scope *tdecs* and permits defined signatures to refer to abstract type components in *sdecs*.

A free occurrence of $\overline{unitid}$ in an IL unit or interface represents a definite reference to that unit, where $\bar{\cdot}$ denotes a function taking unit identifiers to TSIL variables. We assume that this function is injective, that there are countably many variables not in its range, and that when a "fresh" variable is chosen, the choice does not lie in its range. (The same overbar notation is used to represent a function taking TSEL identifiers to TSIL labels; no confusion can result because

$impl$	$:=$	mod	module
$intf$	$::=$	$var : [sdecs]; tdecs$	signature for unit and its top-level declarations
$tdecs$	$::=$	$\cdot$	
		$tdecs, tdec$	
$tdec$	$::=$	$sigid : \mathsf{Sig} = sig$	
Γ	$::=$	$decs$	declarations

Figure 24: Realization of IL syntax for TS

Judgement...	*Meaning...*
$decs \vdash tdecs$ ok	$tdecs$ is well-formed
$decs \vdash tdecs \equiv tdecs'$	$tdecs$ equivalence
$decs \vdash tdecs \supset tdecs'$	$tdecs$ inclusion

Figure 25: Judgements of the IL realization for TS

labels and variables are kept separate in the syntax.)

For example, the source interface

```
open (* empty *) in
    type t

    signature S =
    sig
        type s = t
    end
end
```

corresponds to the compiled interface

$$var : [\overline{\mathtt{t}} \rhd var_t : \Omega];$$
$$\mathtt{S} : \mathsf{Sig} = [\overline{\mathtt{s}} \rhd var_s : \Omega = var.var_t].$$

A compiled unit `A` with this interface defines exactly one (type) component. The source interface `open A in structure X : S end` uses the signature definition and a definite reference to this type component. The corresponding compiled interface is

$$var' : [\overline{\mathtt{X}} \rhd var_X : [\overline{\mathtt{s}} \rhd var_s : \Omega = \overline{\mathtt{A}}.var_t]];$$
$$\cdot.$$

We realize the IL judgemental stubs for TS in Appendix D.1 using the auxiliary judgement forms given in Figure 25. (The stub $\vdash \Gamma$ ok is realized by the TS judgement $\vdash decs$ ok.) The rules build on the TSIL static semantics to type-check the IL.

4.2 Realization of the Elaborator for TS

The TSEL permits higher-order functors but not signature definitions. We change the TSEL and elaborator for compatibility with the TDEL:

- Remove $\mathsf{functor}$ *funbind* from the syntax of TSEL structure declarations [HS97, page 34] and TS rule 205 for elaborating them.

- Remove $\mathsf{functor}\ \mathit{funid}(\mathit{strid} : \mathit{sigexp}) : \mathit{sigexp}'$ from the syntax of TSEL structure specifications and TS rule 224 for elaborating them.

- Extend TS elaboration contexts from structure declaration lists (*sdecs*) to unit declaration lists (*udecs*):

$$\begin{array}{rcl} \mathit{udecs} & ::= & \cdot \\ & & \mathit{udecs}, \mathit{udec} \\ \mathit{udec} & ::= & \mathit{sdec} \\ & & \mathit{tdec}. \end{array}$$

  In a TS elaboration context of the form *sdec*, *udecs*, the scope of the bound variable BV(*sdec*) is *udecs*.

- Add *sigid* to the syntax of TSEL signature expressions, extend the TS elaborator judgment

$$\mathit{udecs} \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig}$$

  with a rule for elaborating them, and extend the TS elaborator with a judgement

$$\mathit{udecs} \vdash_{\mathsf{ctx}} \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}$$

  for signature identifier lookup. (The rules are in Appendix D.2.)

We realize the basis interface for TS with

$$\mathit{intf}_{\mathit{basis}} = \mathit{var} : \mathit{sig}_{\mathit{basis}}; \cdot$$

where $\mathit{sig}_{\mathit{basis}}$ contains at least the following fields which define three exceptions:

$$\begin{array}{ll} [\overline{\mathsf{Bind}}^\star & :[\mathsf{tag}{:}\mathsf{Unit\ Tag}, \overline{\mathsf{Bind}}{:}\mathsf{Tagged}], \\ \overline{\mathsf{Match}}^\star & :[\mathsf{tag}{:}\mathsf{Unit\ Tag}, \overline{\mathsf{Match}}{:}\mathsf{Tagged}], \\ \mathsf{fail}^\star & :[\mathsf{tag}{:}\mathsf{Unit\ Tag}, \mathsf{fail}{:}\mathsf{Tagged}]]. \end{array}$$

This choice ensures that TS elaboration contexts declare a structure $\overline{\mathsf{basis}} : \mathit{sig}_{\mathit{basis}}$, as assumed by the TS elaborator.

We realize the elaborator judgemental stubs for TS in Appendix D.2. Elaboration uses a renaming, σ, to support opening units while elaborating a unit or interface expression. We define the syntax of renamings in Figure 26 and give auxiliary judgement forms in Figure 27. TS elaboration contexts are created by rule 79. A unit with interface $\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}$ is described in *udecs* by the declaration $1 \rhd \overline{\mathit{unitid}} : [\mathit{sdecs}]$ and, if it is opened, by declarations that make its components available for identifier lookup. The declarations that open *unitid* are:

$$1^\star \rhd \mathit{var} : \mathit{sig}, \mathit{tdecs}$$

where *sig* is the selfified signature of $\overline{\mathit{unitid}}$. The declaration of *var* makes the expression, type, structure, and functor components of *unitid* visible and the declarations *tdecs* make its signature components visible. After elaboration, we substitute $\overline{\mathit{unitid}}$ for free occurrences of *var* to obtain HSIL code that does not depend on *var*.

$$\begin{array}{rcl} \sigma & ::= & \cdot \\ & & \sigma, var/var' \end{array}$$

Figure 26: Renamings

Judgement...	*Meaning...*
$udecs \vdash sdecs; tdecs \rightsquigarrow intf : \mathsf{Intf}$	interface creation
$udecs \vdash topdec \rightsquigarrow sbnds : (sdecs; tdecs)$	top-level declaration elaboration
$udecs \vdash topspec \rightsquigarrow sdecs; tdecs$	top-level specification elaboration
$udecs \vdash sigbind \rightsquigarrow tdecs$	signature binding elaboration
$udecs \vdash sigexp \rightsquigarrow sig : \mathsf{Sig}$	signature expression elaboration
$udecs \vdash funspec \rightsquigarrow sdecs$	functor specification elaboration
$udecs \vdash_{\mathsf{ctx}} sigid \rightsquigarrow sig : \mathsf{Sig}$	signature lookup
$adecs \vdash \mathsf{open}\ unitids \rightsquigarrow udecs, \sigma$	*udecs* declares units in *adecs* and top-level identifiers in *unitids*
$\vdash udecs\ \mathsf{ok}$	*udecs* is well-formed
$udecs \vdash decs$	context coercion

Figure 27: Judgements of the elaborator realization for TS

4.3 Realization of the Linker for TS

We realize programs for TS as follows:

$$prog \quad ::= \quad exp : \{\}.$$

A program is a (closed) HSIL expression of type unit. We realize the linking judgemental stubs for TS in Appendix D.3 using the auxiliary judgement form

$$\vdash assm \rightsquigarrow bnds : decs$$

to obtain TDIL bindings for the units in an assembly. The rules for completion build an expression of the form $[sbnds].\mathsf{it}$ where the structure $[sbnds]$ contains a field for each unit in the assembly and a final field $\mathsf{it} = \{\}$ of type unit. The structure for unit *unitid* is $lab_{unitid} \triangleright \overline{unitid} = mod$ where *mod* is supplied by completion for the basis unit and taken from the assembly for all other units.

4.4 Dynamic Semantics of Programs in TS

We use the HSIL dynamic semantics to evaluate programs. Given a well-formed program $prog = exp : \{\}$, we construct an initial state $\Sigma = (\cdot, \cdot, exp)$.

5 TD Realization

After a brief review of *The Definition of Standard ML* [MTHM97], we realize the semantics of separate compilation for TD (Sections 5.1–5.3) and define a dynamic semantics for programs arising from complete assemblies (Section 5.4).

Judgement...	*Meaning...*
$B \vdash \mathit{topdec} \Rightarrow B'$	top-level declaration elaboration
$B \vdash \mathit{strdec} \Rightarrow E$	structure-level declaration elaboration
$B \vdash \mathit{strbind} \Rightarrow SE$	structure binding elaboration
$B \vdash \mathit{strexp} \Rightarrow E$	structure expression elaboration
$B \vdash \mathit{sigdec} \Rightarrow G$	signature declaration elaboration
$B \vdash \mathit{sigbind} \Rightarrow G$	signature binding elaboration
$B \vdash \mathit{sigexp} \Rightarrow E$	signature expression elaboration
$B \vdash \mathit{sigexp} \Rightarrow \Sigma$	
$B \vdash \mathit{spec} \Rightarrow E$	specification elaboration
$B \vdash \mathit{strdesc} \Rightarrow SE$	structure description elaboration
$B \vdash \mathit{fundec} \Rightarrow F$	functor declaration elaboration
$B \vdash \mathit{funbind} \Rightarrow F$	functor binding elaboration
$\Sigma \geq E$	signature instantiation
$\Phi \geq (E, (T)E')$	functor signature instantiation
$E_1 \succ E_2$	signature enrichment
$\vdash A$ ok	A contains well-formed type structures

Figure 28: TD elaboration judgements (summary)

TD defines TDEL through elaboration and evaluation relations between source phrases and a collection of *semantic objects* called the TD internal language (TDIL). The static semantic objects record just enough information for type-checking and the dynamic semantic objects record just enough information for evaluation.

The judgement forms of the TD elaborator are summarized in Figure 28 and the corresponding TDIL objects are summarized in Figure 29. (The judgements for elaborating core and TDEL program constructs—and the corresponding semantic objects—are omitted.) In presenting, and later extending, the TDIL, we use the following notation:

- $\mathrm{Fin}(A)$ denotes the set of finite subsets of A.
- $A \times B$ denotes the cartesian product of A and B and A^k denotes a sequence of length k whose range is a subset of A.
- $\mathrm{Fin}(A)$ denotes the set of finite subsets of A.
- $A \stackrel{\mathrm{fin}}{\to} B$ denotes the set of finite, partial functions from A to B.
- $A \cup B$ denotes the disjoint union of A and B and a/b is a compound metavariable that ranges over this union.

Elaboration judgements have the form $A \vdash \mathit{phrase} \Rightarrow A'$ and mean that *phrase* elaborates to A' in context A. To account for type generativity and type sharing in Standard ML, the rules are state-passing: They track the set of type names that "have been generated" to ensure that "new"

$$
\begin{array}{rcl}
B \text{ or } T, F, G, E & \in & \mathrm{Basis} = \mathrm{TyNameSet} \times \mathrm{FunEnv} \times \mathrm{SigEnv} \times \mathrm{Env} \\
T & \in & \mathrm{TyNameSet} = \mathrm{Fin}(\mathrm{TyName}) \\
F & \in & \mathrm{FunEnv} = \mathrm{FunId} \xrightarrow{\mathrm{fin}} \mathrm{FunSig} \\
G & \in & \mathrm{SigEnv} = \mathrm{SigId} \xrightarrow{\mathrm{fin}} \mathrm{Sig} \\
E \text{ or } (SE, TE, VE) & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
\Phi \text{ or } (T)(E, (T')E') & \in & \mathrm{FunSig} = \mathrm{TyNameSet} \times (\mathrm{Env} \times \mathrm{Sig}) \\
\Sigma \text{ or } (T)E & \in & \mathrm{Sig} = \mathrm{TyNameSet} \times \mathrm{Env} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{\mathrm{fin}} \mathrm{Env} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \xrightarrow{\mathrm{fin}} \mathrm{TyStr} \\
VE & \in & \mathrm{ValEnv} = \mathrm{VId} \xrightarrow{\mathrm{fin}} \mathrm{TypeScheme} \times \mathrm{IdStatus} \\
(\theta, VE) & \in & \mathrm{TyStr} = \mathrm{TypeFcn} \times \mathrm{ValEnv} \\
\sigma \text{ or } \forall\alpha^{(k)}.\tau & \in & \mathrm{TypeScheme} = \bigcup_{k\geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
\theta \text{ or } \Lambda\alpha^{(k)}.\tau & \in & \mathrm{TypeFcn} = \bigcup_{k\geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
\tau & \in & \mathrm{Type} = \mathrm{TyVar} \times \mathrm{RowType} \times \mathrm{FunType} \times \mathrm{ConsType} \\
(\alpha_1, \cdots, \alpha_k) \text{ or } \alpha^{(k)} & \in & \mathrm{TyVar}^k \\
\varrho & \in & \mathrm{RowType} = \mathrm{Lab} \xrightarrow{\mathrm{fin}} \mathrm{Type} \\
\tau \to \tau' & \in & \mathrm{FunType} = \mathrm{Type} \times \mathrm{Type} \\
 & & \mathrm{ConsType} = \bigcup_{k\geq 0} \mathrm{ConsType}^{(k)} \\
\tau^{(k)}t & \in & \mathrm{ConsType}^{(k)} = \mathrm{Type}^k \times \mathrm{TyName}^{(k)} \\
(\tau_1, \cdots, \tau_k) \text{ or } \tau^{(k)} & \in & \mathrm{Type}^k \\
t & \in & \mathrm{TyName} \text{ (type names)} \\
\mathit{funid} & \in & \mathrm{FunId} \text{ (functor identifiers)} \\
\mathit{sigid} & \in & \mathrm{SigId} \text{ (signature identifiers)} \\
\mathit{strid} & \in & \mathrm{StrId} \text{ (structure identifiers)} \\
\mathit{tycon} & \in & \mathrm{TyCon} \text{ (type constructors)} \\
\mathit{vid} & \in & \mathrm{VId} \text{ (value identifiers)} \\
\alpha \text{ or } \mathit{tyvar} & \in & \mathrm{TyVar} \text{ (type variables)} \\
\mathit{is} & \in & \mathrm{IdStatus} = \{\mathsf{c}, \mathsf{e}, \mathsf{v}\} \text{ (identifier status descriptors)} \\
\mathit{lab} & \in & \mathrm{Lab} \text{ (labels)}
\end{array}
$$

Figure 29: TDIL static semantic objects (summary)

Judgement...	*Meaning...*
$s, B \vdash \mathit{topdec} \Rightarrow B', s'$	top-level declaration evaluation
$s, B \vdash \mathit{strdec} \Rightarrow E/p, s'$	structure-level declaration evaluation
$s, B \vdash \mathit{strbind} \Rightarrow SE/p, s'$	structure binding evaluation
$s, B \vdash \mathit{strexp} \Rightarrow E/p, s'$	structure expression evaluation
$s, B \vdash \mathit{fundec} \Rightarrow F, s'$	functor declaration evaluation
$s, B \vdash \mathit{funbind} \Rightarrow F, s'$	functor binding evaluation
$\mathit{IB} \vdash \mathit{sigdec} \Rightarrow G$	signature declaration elaboration
$\mathit{IB} \vdash \mathit{sigbind} \Rightarrow G$	signature binding elaboration
$\mathit{IB} \vdash \mathit{sigexp} \Rightarrow I$	signature expression elaboration
$\mathit{IB} \vdash \mathit{sigexp} \Rightarrow \Sigma$	
$\mathit{IB} \vdash \mathit{spec} \Rightarrow I$	specification elaboration
$\mathit{IB} \vdash \mathit{strdesc} \Rightarrow \mathit{SI}$	structure description elaboration
$E \downarrow I = E'$	signature ascription

Figure 30: TD evaluation judgements (summary)

type names can always be chosen to represent abstract types in *phrase*. The type names bound in A are those that were generated prior to elaborating *phrase*, the type names bound in A' are those that are generated by elaborating *phrase*, and the type names free in A' and bound in A represent references in *phrase* to "old" types.

For example, in a basis $B = T, F, G, E$, the set T binds type names with scope F, G, E. In the judgement

$$B \vdash \mathit{topdec} \Rightarrow B',$$

B describes everything that was elaborated prior to *topdec* and B' describes the components of *topdec*. Abstract types declared in *topdec* are represented by bound type names in B'.

The instantiation and enrichment relations describe signature matching in terms of TDIL environments and signatures. In a signature $\Sigma = (T)E$, the set T binds type names with scope E that represent abstract types specified by the signature. Instantiation $(T)E_1 \geq E_2$ checks that E_2 can be obtained from E_1 by substituting for type names in T. Enrichment $E_1 \succ E_2$ permits E_1 to have more components than E_2 and for components to be less polymorphic. An environment E matches Σ if there exists E' such that $\Sigma \geq E' \prec E$.

The TD dynamic semantics is a big-step, call-by-value operational semantics. The judgement forms for TD evaluation are summarized in Figure 30 and the corresponding TDIL objects are given in Figure 31.[7] (The judgements for evaluating core and TDEL program constructs are omitted.) Evaluation judgements have the form $s, A \vdash \mathit{phrase} \Rightarrow A', s'$ and mean that *phrase* evaluates to A' in context A, where s and s' are states before and after evaluation. Most TDEL type information is erased prior to evaluation but signature ascriptions are retained. The rules for signature ascription use $E \downarrow I$ to thin the environment E so that a subsequent evaluation of `open` does not shadow

[7]In many cases, the same names are used for static and dynamic TDIL objects. Such names refer to static semantic objects except in Section 5.4 and Appendix E.4 where they refer to dynamic semantic objects unless the subscript $(\cdot)_{\mathrm{STAT}}$ is used.

$$
\begin{array}{rcl}
(F, G, E) \text{ or } B & \in & \mathrm{Basis} = \mathrm{FunEnv} \times \mathrm{SigEnv} \times \mathrm{Env} \\
(G, I) \text{ or } IB & \in & \mathrm{IntBasis} = \mathrm{SigEnv} \times \mathrm{Int} \\
(mem, ens) \text{ or } s & \in & \mathrm{State} = \mathrm{Mem} \times \mathrm{ExNameSet} \\
[e] \text{ or } p & \in & \mathrm{Pack} = \mathrm{ExVal} \\
F & \in & \mathrm{FunEnv} = \mathrm{FunId} \xrightarrow{\mathrm{fin}} \mathrm{FunctorClosure} \\
G & \in & \mathrm{SigEnv} = \mathrm{SigId} \xrightarrow{\mathrm{fin}} \mathrm{Int} \\
(SE, TE, VE) \text{ or } E & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
(SI, TI, VI) \text{ or } I & \in & \mathrm{Int} = \mathrm{StrInt} \times \mathrm{TyInt} \times \mathrm{ValInt} \\
mem & \in & \mathrm{Mem} = \mathrm{Addr} \xrightarrow{\mathrm{fin}} \mathrm{Val} \\
ens & \in & \mathrm{ExNameSet} = \mathrm{Fin}(\mathrm{ExName}) \\
e & \in & \mathrm{ExVal} = \mathrm{ExName} \cup (\mathrm{ExName} \times \mathrm{Val}) \\
(strid : I, strexp, B) & \in & \mathrm{FunctorClosure} = (\mathrm{StrId} \times \mathrm{Int}) \times \mathrm{StrExp} \times \mathrm{Basis} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{\mathrm{fin}} \mathrm{Env} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \xrightarrow{\mathrm{fin}} \mathrm{ValEnv} \\
VE & \in & \mathrm{ValEnv} = \mathrm{VId} \xrightarrow{\mathrm{fin}} \mathrm{Val} \times \mathrm{IdStatus} \\
SI & \in & \mathrm{StrInt} = \mathrm{StrId} \xrightarrow{\mathrm{fin}} \mathrm{Int} \\
TI & \in & \mathrm{TyInt} = \mathrm{TyCon} \xrightarrow{\mathrm{fin}} \mathrm{ValInt} \\
VI & \in & \mathrm{ValInt} = \mathrm{VId} \xrightarrow{\mathrm{fin}} \mathrm{IdStatus} \\
v & \in & \mathrm{Val} = \ \{:=\} \cup \mathrm{SVal} \cup \mathrm{BasVal} \cup \mathrm{VId} \\
 & & \qquad\quad \cup\, (\mathrm{VId} \times \mathrm{Val}) \cup \mathrm{ExVal} \\
 & & \qquad\quad \cup\, \mathrm{Record} \cup \mathrm{Addr} \cup \mathrm{FcnClosure} \\
r & \in & \mathrm{Record} = \mathrm{Lab} \xrightarrow{\mathrm{fin}} \mathrm{Val} \\
(match, E, VE) & \in & \mathrm{FcnClosure} = \mathrm{Match} \times \mathrm{Env} \times \mathrm{ValEnv} \\
en & \in & \mathrm{ExName} \text{ (exception names)} \\
a & \in & \mathrm{Addr} \text{ (addresses)} \\
sv & \in & \mathrm{SVal} \text{ (special values)} \\
b & \in & \mathrm{BasVal} \text{ (basic values)}
\end{array}
$$

Figure 31: TDIL dynamic semantic objects

identifiers bound in E but hidden by I. The dynamic semantics elaborates signatures rather than compute I from the TDIL objects generated by "full" elaboration.

5.1 Realization of the Internal Language for TD

To account for type sharing with separate compilation, we assume that the TD elaborator generates *principal* TDIL and we ensure that the TD realization preserves principality. For example, we assume that the TD elaborator judgement

$$B \vdash \mathit{sigexp} \Rightarrow \Sigma$$

produces a signature Σ that is principal for *sigexp* in B, meaning that the type names in Σ share only when required by the source. Without principality, it would be possible for separately elaborated assemblies to use the same type name t for different types or to use distinct type names t and t' for the same type so that linking them together would introduce "accidental" sharing or would fail to impose required sharing.

We distinguish between *external* and *internal* names for types. An internal name t is a TDIL type name. An external name *path* is used to make definite reference to an externally defined type. A path of the form *unitid.longtycon* refers to the type constructor *longtycon* defined in the unit *unitid*. A path of the form *unitid.n* refers to a type defined in the unit *unitid* that, in the source for *unitid*, escapes its scope. (Labels n are assigned to such types when interfaces are inferred.) The rules avoid accidental sharing by alpha-varying bound internal names when interfaces are added to context and preserve required sharing between units and assemblies by using external names in interfaces.

We extend the TDIL in Figure 32 and realize the IL syntactic stubs for TD in Figure 33. A compiled unit contains source code and a record of interface ascriptions. A compiled interface comprises a basis B describing units with this interface, imports IP governing external references to types, and labels L assigning external names to those types bound in B that can not be named in source code. In an interface IP, B, L with $B = T, F, G, E$, the set $\mathrm{dom}(IP)$ binds type names with scope B and the set T binds type names with scope F, G, E, and L. A well-formed interface $IP, (T, F, G, E), L$ has no free type names and satisfies $\mathrm{rng}(L) \subset T$. A context UE maps a unit identifier to the basis B and labels L describing it, with appropriate sharing.

We realize the IL judgemental stubs for TD in Appendix E.1 using the auxiliary judgement forms given in Figure 34. An elaboration basis B binds all of the type names generated by units in the assembly and the top-level components of any opened units. A compiled unit is well-formed if its top-level declaration elaborates and any interface ascriptions respect the sub-interface relation. The sub-interface relation, analogous to signature matching, relies on instantiation and enrichment.

$$\begin{array}{rcl}
IP & \in & \mathrm{Imports} = \mathrm{TyName} \xrightarrow{\mathrm{fin}} \mathrm{Path} \\
L & \in & \mathrm{Labels} = \mathrm{Nat} \xrightarrow{\mathrm{fin}} \mathrm{TyName} \\
IE & \in & \mathrm{ImportEnv} = \mathrm{Path} \xrightarrow{\mathrm{fin}} \mathrm{TyName} \\
UE & \in & \mathrm{UnitEnv} = \mathrm{UnitId} \xrightarrow{\mathrm{fin}} \mathrm{Basis} \times \mathrm{Labels} \\
\mathit{path} & \in & \mathrm{Path} = \mathrm{UnitId} \times (\mathrm{LongTyCon} \cup \mathrm{Nat}) \\
\mathit{unitid} & \in & \mathrm{UnitId}\ (\text{unit identifiers}) \\
n & \in & \mathrm{Nat}\ (\text{natural numbers})
\end{array}$$

Figure 32: TDIL extensions for the IL

intf	::=	IP, B, L	imports, basis, and labels
impl	::=	$\mathit{unitexp}$	basic
		$\mathit{impl} : \mathit{intf}$	coerced to intf
Γ	::=	UE	unit environment

Figure 33: Realization of IL syntax for TD

Judgement...	*Meaning...*
$\Gamma \vdash B \Rightarrow \mathit{intf} : \mathsf{Intf}$	interface creation
$\Gamma \vdash \mathit{intf} \Rightarrow B, L$	interface realization
$\Gamma \vdash \mathsf{open}\ \mathit{unitids} \Rightarrow B$	B declares type names in Γ and top-level identifiers in $\mathit{unitids}$

Figure 34: Judgements of the IL realization for TD

5.2 Realization of the Elaborator for TD

We realize the basis interface for TD with

$$\mathit{intf}_{\mathit{basis}} = \{\}, B_0, \{\}$$

where B_0 is defined in [MTHM97, Appendix C]. This choice ensures that every TD elaboration basis B declares the types, values, and exceptions assumed by the TD elaborator and derived forms.

We realize the elaboration judgemental stubs for TD in Appendix E.2 using the auxiliary judgement forms given in Figure 35. Rule 110 for interface ascription produces a compiled unit of the form $\mathit{impl}_0 : \mathit{intf}$. During evaluation, the basis for such a unit is thinned analogous to the treatment of signature ascription in the TD evaluator.

5.3 Realization of the Linker for TD

We realize programs for TD as follows:

$$\mathit{prog} \quad ::= \quad \mathit{assm}.$$

A program is a (complete) IL assembly. We realize the linking judgemental stubs for TD in Appendix E.3.

5.4 Dynamic Semantics of Programs in TD

The dynamic semantics of programs is based on the dynamic semantics for TDEL, on the dynamic TDIL extended with

$$\mathit{UE} \quad \in \quad \mathrm{UnitEnv} = \mathrm{UnitId} \xrightarrow{\mathrm{fin}} \mathrm{Basis},$$

and on the basis B_0 and state s_0 defined in [MTHM97, Appendix D].

Judgement...	*Meaning...*
$B \vdash \mathit{topspec} \Rightarrow B'$	top-level specification elaboration
$B \vdash \mathit{funspec} \Rightarrow F$	functor specification elaboration

Figure 35: Judgements of the elaborator realization for TD

Judgement...	*Meaning...*
$\vdash prog \Rightarrow UE/p, s$	program evaluation
$s, UE \vdash assm \Rightarrow UE'/p, s'$	assembly evaluation
$s, UE \vdash impl \Rightarrow B/p, s'$	unit evaluation
$UE \vdash \mathsf{open}\ unitids \Rightarrow B$	B binds top-level identifiers in *unitids*

Figure 36: Judgements of the dynamic semantics of programs in the TD realization

The judgement forms for evaluating programs are given in Figure 36 and the rules are given in Appendix E.4. The rules evaluate the units in a program in sequence, stopping on uncaught exceptions. Rules 119 and 120 implement the basis unit.

6 Related Work

A distinction between this proposal and most other languages for separate compilation is that the EL is stratified into three levels (SML core, SML modules, and separate compilation) rather than two or one. Pragmatically, this ensure that the proposal is compatible with existing SML code and compilers. The IL is similarly stratified because it is unclear how to extend the type theory for ML modules in [Ler94, HL94] to account for signature definitions in structures.

A second distinction is that EL and IL units and interfaces are not independently meaningful, but instead contain free identifiers whose types are obtained from the ambient assembly. This makes source and compiled interfaces smaller and is natural given our use of definite references.

In this proposal, we take the view that a library is an assembly that can be linked with other assemblies. The benefits of this approach are its simplicity and its support for selective linking. We provide no mechanism for managing the global namespace of unit identifiers so the names of "private" library units may interfere with names used by other assemblies. We leave the solution of this namespace problem to future work.

We have presented the semantics of separate compilation in a form that is largely independent of the underlying semantic framework for SML. Modular presentations of this sort are not new. Ancona and Zucca [AZ02] define their module system over an unspecified core language, using explicit substitutions to represent core terms that refer to modules. Leroy [Ler00] implements the type-theory for ML modules in a way that is parameterized by a core language and its type-checker. He instantiates the system with two core languages, mini-ML and mini-C.

Languages for Separate Compilation. Cardelli [Car97] investigates separate compilation for the simply-typed λ-calculus and discusses some of the obstacles to overcome in designing a language for separate compilation. Several specific aspects of the current design arise in this simpler setting, including the use of interfaces to govern separate type-checking and type-safe linking and the use of globally unique names so that linking can resolve external references. Glew and Morrisett [GM99] describe separate compilation for Typed Assembly Language [MWCG99]. Their language, MTAL, permits type definitions, abstract types, and polymorphic types in interfaces and supports recursive linking. They suggest an explicit α-conversion operation that turns a global name defined by a typed object file into a local one to alleviate the problem with global scoping.

Harper and Pierce [HP05] discuss language design for advanced modularity mechanisms, including separate compilation. Particularly relevant to the current work is their discussion of abstract type components and type sharing. They describe the use of definite references to avoid the coherence problems (and excess sharing specifications) that arise from aliasing. They also discuss side-effects and the important distinction between initialization and interface dependencies.

Mixin modules are incomplete and mutually recursive code fragments that can be separately compiled and flexibly linked together. Mixin systems have been used or proposed for Standard ML [DS96], scheme [FF98], and C [RFS$^+$00]. Mixin calculi [WV00, AZ02] are expressive enough to encode various λ-calculi, object calculi, and module systems. Call-by-value mixin calculi have been defined by translation to a λ-calculus and by a small-step reduction semantics [HLW04]. A type system for simply-typed mixins with principal typings and a compositional type inference algorithm has been developed and extended with ML-style let-polymorphism [MW05]. These calculi do not support ML-style type components, abstract types, and type sharing, but Ancona and Zucca [AZ02] suggest how their calculus could be extended to support type components in a way that respects the phase distinction.

Objective CAML. The separate compilation system implemented as part of Objective CAML has some important similarities to the design presented here. Objective CAML [LDG$^+$05] provides notions of units (`.ml` files) and interfaces (`.mli` files) and, as here, a unit is coerced to its stated interface when one is provided.

There are also at least two important differences. First, Objective CAML is defined by its implementation and related tools, rather than by a formal specification. Second, like many systems but unlike the design presented here, Objective CAML obtains the name of a unit from the name of the file that contains it. Consequently, the selection of unit names is limited by file system considerations, and restructuring of a project on its storage device must be accompanied by changes to the code.

Moscow ML. The separate compilation system implemented as part of Moscow ML [RRS00] is similar to that in Objective CAML, with one notable extension. A programmer may describe the units, interfaces, and dependencies in a program in a form that is similar to an EL assembly. The `mosmake` [Mak02] tool converts such a description to a makefile.

Standard ML of New Jersey. The Compilation Manager for Standard ML of New Jersey (CM) [Blu02] is a convenient tool for compiling whole SML programs. CM permits a program to be divided into a hierarchy of libraries [BA99]. A library comprises a list of imported libraries, SML source files, and a list of SML symbols exported by the library. Dependencies between libraries are explicit but dependencies among the SML source files in a library are inferred [Blu99, HLPR94]. CM can generate SML source using tools or shell recipes, provides control over the SML identifiers visible to an SML file, and supports conditional compilation, parallel compilation, and cut-off incremental recompilation. CM has no notion akin to an EL interface and does not support compiling a unit with an ascribed interface or compiling against an unimplemented unit described by an interface. A tool to translate a web of interconnected CM files to a complete EL assembly would enable users to compile programs written in CM notation with implementations of the proposed separate compilation facility.

Acknowledgements

We thank Dan Licata for comments on a draft of this manuscript.

References

[ATW94] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[AZ02] Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

[BA99] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847, 1999.

[Blu99] Matthias Blume. Dependency analysis for Standard ML. *ACM Transactions on Programming Languages and Systems*, 21(4):790–812, 1999.

[Blu02] Matthias Blume. CM: The SML/NJ compilation and library manager (for SML/NJ version 110.40 and later) user manual, 2002. `http://www.smlnj.org/doc/CM/new.pdf`.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277. ACM Press, 1997.

[DS96] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, 1996.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

[GM99] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261. ACM Press, 1999.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137. ACM Press, 1994.

[HLPR94] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental recompilation for Standard ML of New Jersey. Technical Report CMU-CS-94-116, School of Computer Science, Carnegie Mellon University, February 1994.

[HLW04] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *European Symposium on Programming*, pages 64–78. Springer-Verlag LNCS 2986, 2004.

[HP05] Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–346. MIT Press, 2005.

[HS97] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, School of Computer Science, Carnegie Mellon University, June 1997.

[HS00] Robert Harper and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[LDG+05] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.09: Documentation and user's manual, 2005. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[Mak02] Henning Makholm. Mosmake version 0.9, November 2002. `http://www.diku.dk/~makholm/mosmake/`.

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

[MW05] Henning Makholm and J. B. Wells. Type inference, principal typings, and let-polymorphism for first-class mixin modules. In *Proceedings of the ACM/SIGPLAN International Conference on Functional Programming*, pages 156–167. ACM Press, 2005.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[RFS+00] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–3660. Usenix Association, 2000.

[RRS00] Sergei Romanenko, Claudio Russo, and Peter Sestoft. Moscow ML owner's manual version 2.00, June 2000. `http://www.dina.kvl.dk/~sestoft/mosml/manual.pdf`.

[TIL] TILT web site. `http://www.tilt.cs.cmu.edu/`.

[WV00] J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In *Proceedings of the European Symposium on Programming Languages and Systems*, pages 412–428. Springer-Verlag LNCS 1782, 2000.

A Internal Language Static Semantics

In the inference rules, either all optional elements or none must be present.

Definition 1. *The domain of an IL context,* $\mathrm{dom}(adecs)$*, is defined by*

$$\mathrm{dom}(adec_1, \ldots, adec_n) = \{\mathrm{dom}(adec_1), \ldots, \mathrm{dom}(adec_n)\}$$

where $\mathrm{dom}(adec)$ *is defined by* $\mathrm{dom}(unitid : intf) = unitid$*.*

$$\boxed{adecs \vdash assm\ \mathsf{ok}}$$

$$\frac{\vdash adecs\ \mathsf{ok}}{adecs \vdash \cdot\ \mathsf{ok}} \tag{1}$$

$$\frac{\begin{array}{c} unitid \notin \mathrm{dom}(adecs) \\ adecs \vdash intf : \mathsf{Intf} \\ adecs, unitid : intf \vdash assm\ \mathsf{ok} \end{array}}{adecs \vdash unitid : intf, assm\ \mathsf{ok}} \tag{2}$$

$$\frac{\begin{array}{c} unitid \notin \mathrm{dom}(adecs) \\ adecs \vdash unite : intf \\ adecs, unitid : intf \vdash assm\ \mathsf{ok} \end{array}}{adecs \vdash unitid : intf = unite, assm\ \mathsf{ok}} \tag{3}$$

$$\boxed{adecs \vdash intf : \mathsf{Intf}}$$

$$\frac{\begin{array}{c} adecs \vdash \Gamma \\ \Gamma \vdash intf : \mathsf{Intf} \end{array}}{adecs \vdash intf : \mathsf{Intf}} \tag{4}$$

$$\boxed{adecs \vdash unite : intf}$$

$$\frac{\begin{array}{c} unite = \langle \mathsf{internal} \rangle\ \mathsf{require}\ unitid_1 \cdots unitid_n\ \mathsf{in}\ impl \\ unitid_1 \in \mathrm{dom}(adecs) \quad \cdots \quad unitid_n \in \mathrm{dom}(adecs) \\ adecs \vdash impl : intf \end{array}}{adecs \vdash unite : intf} \tag{5}$$

$$\boxed{adecs \vdash impl : intf}$$

$$\frac{\begin{array}{c} adecs \vdash \Gamma \\ \Gamma \vdash impl : intf \end{array}}{adecs \vdash impl : intf} \tag{6}$$

$$\boxed{adecs \vdash intf \equiv intf' : \mathsf{Intf}}$$

$$\frac{adecs \vdash \Gamma \qquad \Gamma \vdash intf \equiv intf' : \mathsf{Intf}}{adecs \vdash intf \equiv intf' : \mathsf{Intf}} \tag{7}$$

$$\boxed{adecs \vdash intf \leq intf' : \mathsf{Intf}}$$

$$\frac{adecs \vdash \Gamma \qquad \Gamma \vdash intf \leq intf' : \mathsf{Intf}}{adecs \vdash intf \leq intf' : \mathsf{Intf}} \tag{8}$$

$$\boxed{\vdash adecs\ \mathsf{ok}}$$

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \tag{9}$$

$$\frac{\vdash adecs\ \mathsf{ok} \qquad unitid \notin \mathrm{dom}(adecs) \qquad adecs \vdash intf : \mathsf{Intf}}{\vdash adecs, unitid : intf\ \mathsf{ok}} \tag{10}$$

B Elaboration

Definition 2. *The domain of an elaboration context,* dom(*edecs*)*, is defined by:*

$$\begin{array}{lcl} \mathrm{dom}(\cdot) & = & \emptyset \\ \mathrm{dom}(edecs, adec) & = & \mathrm{dom}(edecs) \cup \{\mathrm{dom}(adec)\} \\ \mathrm{dom}(edecs, intid : \mathsf{Intf} = intf) & = & \mathrm{dom}(edecs) \cup \{intid\}. \end{array}$$

$$\boxed{\vdash assembly \leadsto assm; edecs}$$

$$\frac{}{\vdash \cdot \leadsto \mathsf{basis} : intf_{basis}; \mathsf{basis} : intf_{basis}} \tag{11}$$

Rule 11: The basis unit is implicit in every EL assembly.

$$\frac{\vdash assembly \leadsto assm; edecs \qquad intid \notin \mathrm{dom}(edecs) \qquad edecs \vdash intexp \leadsto intf}{\vdash assembly, intid = intexp \leadsto assm; edecs, intid : \mathsf{Intf} = intf} \tag{12}$$

$$\frac{\vdash assembly \leadsto assm; edecs \qquad unitid \notin \mathrm{dom}(edecs) \qquad edecs \vdash intexp \leadsto intf}{\vdash assembly, unitid : intexp \leadsto assm, unitid : intf; edecs, unitid : intf} \tag{13}$$

$$\frac{\begin{array}{c}\vdash \mathit{assembly} \leadsto \mathit{assm}; \mathit{edecs} \\ \mathit{unitid} \notin \mathrm{dom}(\mathit{edecs}) \\ \mathit{edecs} \vdash \mathit{unitexp} \leadsto \mathit{unite} : \mathit{intf}\end{array}}{\begin{array}{l}\vdash \mathit{assembly}, \mathit{unitid} = \mathit{unitexp} \leadsto \\ \quad \mathit{assm}, \mathit{unitid} : \mathit{intf} = \mathit{unite}; \mathit{edecs}, \mathit{unitid} : \mathit{intf}\end{array}} \tag{14}$$

$$\frac{\begin{array}{c}\vdash \mathit{assembly} \leadsto \mathit{assm}; \mathit{edecs} \\ \mathit{unitid} \notin \mathrm{dom}(\mathit{edecs}) \\ \mathit{edecs} \vdash \mathit{intexp} \leadsto \mathit{intf} \\ \mathit{edecs} \vdash \mathit{unitexp} \leadsto \mathit{unite}_0 : \mathit{intf}_0 \\ \mathit{edecs} \vdash \mathit{unite}_0 : \mathit{intf}_0 \preceq \mathit{intf} \leadsto \mathit{unite}\end{array}}{\begin{array}{l}\vdash \mathit{assembly}, \mathit{unitid} : \mathit{intexp} = \mathit{unitexp} \leadsto \\ \quad (\mathit{assm}, \mathit{unitid} : \mathit{intf} = \mathit{unite}); (\mathit{edecs}, \mathit{unitid} : \mathit{intf})\end{array}} \tag{15}$$

$$\boxed{\mathit{edecs} \vdash \mathit{unitexp} \leadsto \mathit{unite} : \mathit{intf}}$$

$$\frac{\begin{array}{c}\mathit{edecs} \vdash \mathit{adecs} \\ \mathit{unitexp} = \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \\ \mathit{adecs} \vdash \mathsf{open}\ (\mathsf{basis}\ \mathit{unitids})\ \mathsf{in}\ \mathit{topdec} \leadsto \mathit{impl} : \mathit{intf} \\ \mathit{unite} = \mathsf{internal\ require}\ (\mathsf{basis}\ \mathit{unitids})\ \mathsf{in}\ \mathit{impl}\end{array}}{\mathit{edecs} \vdash \mathit{unitexp} \leadsto \mathit{unite} : \mathit{intf}} \tag{16}$$

Rule 16: The basis unit is implicitly imported for the elaboration of every top-level declaration.

$$\boxed{\mathit{edecs} \vdash \mathit{intexp} \leadsto \mathit{intf} : \mathsf{Intf}}$$

$$\frac{\mathit{edecs} \vdash \mathit{adecs} \quad \mathit{adecs} \vdash \mathsf{open}\ (\mathsf{basis}\ \mathit{unitids})\ \mathsf{in}\ \mathit{topspec} \leadsto \mathit{intf}}{\mathit{edecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topspec} \leadsto \mathit{intf} : \mathsf{Intf}} \tag{17}$$

Rule 17: The basis unit is implicitly imported for the elaboration of every top-level specification.

$$\frac{}{\mathit{edecs}', \mathit{intid} : \mathsf{Intf} = \mathit{intf}, \mathit{edecs}'' \vdash \mathit{intid} \leadsto \mathit{intf} : \mathsf{Intf}} \tag{18}$$

$$\boxed{\mathit{edecs} \vdash \mathit{unite}_0 : \mathit{intf}_0 \preceq \mathit{intf} \leadsto \mathit{unite}}$$

$$\frac{\begin{array}{c}\mathit{unite}_0 = \langle \mathsf{internal} \rangle\ \mathsf{require}\ \mathit{unitids}\ \mathsf{in}\ \mathit{impl}_0 \\ \mathit{edecs} \vdash \mathit{adecs} \quad \mathit{adecs} \vdash \Gamma \quad \Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \leadsto \mathit{impl} \\ \mathit{unite} = \mathsf{require}\ \mathit{unitids}\ \mathsf{in}\ \mathit{impl}\end{array}}{\mathit{edecs} \vdash \mathit{unite}_0 : \mathit{intf}_0 \preceq \mathit{intf} \leadsto \mathit{unite}} \tag{19}$$

$$\boxed{\mathit{edecs} \vdash \mathit{adecs}}$$

$$\frac{}{\cdot \vdash \cdot} \tag{20}$$

$$\frac{\mathit{edecs} \vdash \mathit{adecs}}{\mathit{edecs}, \mathit{unitid} : \mathit{intf} \vdash \mathit{adecs}, \mathit{unitid} : \mathit{intf}} \tag{21}$$

$$\frac{\mathit{edecs} \vdash \mathit{adecs}}{\mathit{edecs}, \mathit{intid} : \mathsf{Intf} = \mathit{intf} \vdash \mathit{adecs}} \tag{22}$$

$$\boxed{\vdash \mathit{edecs}\ \mathsf{ok}}$$

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \tag{23}$$

$$\frac{\vdash \mathit{edecs}\ \mathsf{ok} \quad \mathit{unitid} \notin \mathrm{dom}(\mathit{edecs}) \quad \mathit{edecs} \vdash \mathit{adecs} \quad \mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}}{\vdash \mathit{edecs}, \mathit{unitid} : \mathit{intf}\ \mathsf{ok}} \tag{24}$$

$$\frac{\vdash \mathit{edecs}\ \mathsf{ok} \quad \mathit{intid} \notin \mathrm{dom}(\mathit{edecs}) \quad \mathit{edecs} \vdash \mathit{adecs} \quad \mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}}{\vdash \mathit{edecs}, \mathit{intid} : \mathsf{Intf} = \mathit{intf}\ \mathsf{ok}} \tag{25}$$

C Linking and Completion

We denote by $U(\mathit{assm})$ the function that coerces an IL assembly to an assembly context by dropping unit implementations.

Definition 3. *The domain of an IL assembly,* $\mathrm{dom}(\mathit{assm})$, *is defined by:*

$$\mathrm{dom}(\mathit{assm}) = \mathrm{dom}(U(\mathit{assm})).$$

Definition 4. *The domain of a combination context,* $\mathrm{dom}(\mathit{cdecs})$, *is defined by:*

$$\begin{array}{lcl} \mathrm{dom}(\cdot) & = & \emptyset \\ \mathrm{dom}(\mathit{cdecs}, \mathit{unitid} :_{\langle i \rangle} \mathit{intf}) & = & \mathrm{dom}(\mathit{cdecs}) \cup \{\mathit{unitid}\}. \end{array}$$

$$\boxed{\vdash \mathit{lscript} \rightsquigarrow \mathit{assm}}$$

$$\frac{\vdash \mathit{assms} \rightsquigarrow \mathit{assm}}{\vdash \mathsf{combine}\ \mathit{assms} \rightsquigarrow \mathit{assm}} \tag{26}$$

$$\frac{\begin{array}{c}\vdash \mathit{assms} \rightsquigarrow \mathit{assm}' \\ \mathit{deps} = \mathit{assm}'; \mathit{unitids} \\ \vdash_{\mathit{deps}} \mathit{assm}' \rightsquigarrow \mathit{assm}\end{array}}{\vdash \mathsf{from}\ \mathit{assms}\ \mathsf{select}\ \mathit{unitids} \rightsquigarrow \mathit{assm}} \tag{27}$$

$$\boxed{cdecs \vdash assms \rightsquigarrow assm}$$

$$\frac{}{cdecs \vdash \cdot \rightsquigarrow \cdot} \tag{28}$$

$$\frac{cdecs \vdash assms \rightsquigarrow assm}{cdecs \vdash \cdot; assms \rightsquigarrow assm} \tag{29}$$

$$\frac{\begin{array}{c} unitid \notin \mathrm{dom}(cdecs) \\ cdecs, unitid : intf \vdash assm'; assms \rightsquigarrow assm \end{array}}{cdecs \vdash (unitid : intf, assm'); assms \rightsquigarrow unitid : intf, assm} \tag{30}$$

$$\frac{\begin{array}{c} unitid \notin \mathrm{dom}(cdecs) \\ unite = \langle \mathsf{internal} \rangle \ \mathsf{require}\ unitids\ \mathsf{in}\ impl \\ cdecs, unitid :_{\langle i \rangle} intf \vdash assm'; assms \rightsquigarrow assm \end{array}}{\begin{array}{l} cdecs \vdash (unitid : intf = unite, assm'); assms \rightsquigarrow \\ \quad unitid : intf = unite, assm \end{array}} \tag{31}$$

$$\frac{\begin{array}{c} cdecs = cdecs', unitid : intf', cdecs'' \\ cdecs \vdash adecs \quad adecs \vdash intf \equiv intf' : \mathsf{Intf} \\ cdecs \vdash assm'; assms \rightsquigarrow assm \end{array}}{cdecs \vdash (unitid : intf, assm'); assms \rightsquigarrow assm} \tag{32}$$

$$\boxed{adecs \vdash_{deps} assm \rightsquigarrow assm'}$$

$$\frac{}{adecs \vdash_{deps} \cdot \rightsquigarrow \cdot} \tag{33}$$

$$\frac{\begin{array}{c} \nvdash deps\ \mathsf{requires}\ unitid \\ adecs, unitid : intf \vdash_{deps} assm \rightsquigarrow assm' \end{array}}{adecs \vdash_{deps} unitid : intf\ \langle = unite \rangle, assm \rightsquigarrow assm'} \tag{34}$$

$$\frac{\begin{array}{c} \vdash deps\ \mathsf{requires}\ unitid \\ adecs, unitid : intf \vdash_{deps} assm \rightsquigarrow assm' \end{array}}{\begin{array}{l} adecs \vdash_{deps} unitid : intf\ \langle = unite \rangle, assm \rightsquigarrow \\ \quad unitid : intf\ \langle = unite \rangle, assm' \end{array}} \tag{35}$$

$$\boxed{\vdash deps\ \mathsf{requires}\ unitid}$$

$$\frac{unitid \in \{unitid_1, \ldots, unitid_n\}}{\vdash assm; unitid_1 \cdots unitid_n\ \mathsf{requires}\ unitid} \tag{36}$$

$$\frac{\begin{array}{c} \vdash assm; unitids\ \mathsf{requires}\ unitid' \\ assm = (assm', unitid' : intf\ \langle = unite \rangle, assm'') \\ \vdash intf\ \mathsf{requires}\ unitid \end{array}}{\vdash assm; unitids\ \mathsf{requires}\ unitid} \tag{37}$$

$$\frac{\begin{array}{c}\vdash \mathit{assm}; \mathit{unitids}\ \textsf{requires}\ \mathit{unitid}' \\ \mathit{assm} = (\mathit{assm}', \mathit{unitid}' : \mathit{intf} = \mathit{unite}, \mathit{assm}'') \\ \mathit{unite} = \langle\textsf{internal}\rangle\ \textsf{require}\ \mathit{unitids}\ \textsf{in}\ \mathit{impl} \\ \vdash \mathit{impl}\ \textsf{requires}\ \mathit{unitid}\end{array}}{\vdash \mathit{assm}; \mathit{unitids}\ \textsf{requires}\ \mathit{unitid}} \tag{38}$$

$$\frac{\begin{array}{c}\vdash \mathit{assm}; \mathit{unitids}\ \textsf{requires}\ \mathit{unitid}' \\ \mathit{assm} = (\mathit{assm}', \mathit{unitid}' : \mathit{intf} = \mathit{unite}, \mathit{assm}'') \\ \mathit{unite} = \langle\textsf{internal}\rangle\ \textsf{require}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \textsf{in}\ \mathit{impl} \\ \mathit{unitid} \in \{\mathit{unitid}_1, \ldots, \mathit{unitid}_n\}\end{array}}{\vdash \mathit{assm}; \mathit{unitids}\ \textsf{requires}\ \mathit{unitid}} \tag{39}$$

$$\boxed{\mathit{adecs} \vdash \mathit{assm}\ \textsf{complete}}$$

$$\frac{\vdash \mathit{adecs}\ \textsf{ok}}{\mathit{adecs} \vdash \cdot\ \textsf{complete}} \tag{40}$$

$$\frac{\begin{array}{c}\textsf{basis} \notin \mathrm{dom}(\mathit{adecs}) \\ \mathit{adecs} \vdash \mathit{intf} \equiv \mathit{intf}_{\mathit{basis}} : \textsf{Intf} \\ \mathit{adecs}, \textsf{basis} : \mathit{intf} \vdash \mathit{assm}\ \textsf{complete}\end{array}}{\mathit{adecs} \vdash \textsf{basis} : \mathit{intf}, \mathit{assm}\ \textsf{complete}} \tag{41}$$

Rule 41: The basis unit is the only unit that may be unimplemented in a complete IL assembly. Conceptually, the judgement $\vdash \mathit{assm} \rightsquigarrow \mathit{prog}$ supplies an implementation.

$$\frac{\begin{array}{c}\mathit{unitid} \notin \mathrm{dom}(\mathit{adecs}) \cup \{\textsf{basis}\} \\ \mathit{adecs} \vdash \mathit{unite} : \mathit{intf} \\ \mathit{adecs}, \mathit{unitid} : \mathit{intf} \vdash \mathit{assm}\ \textsf{complete}\end{array}}{\mathit{adecs} \vdash \mathit{unitid} : \mathit{intf} = \mathit{unite}, \mathit{assm}\ \textsf{complete}} \tag{42}$$

$$\boxed{\mathit{cdecs} \vdash \mathit{adecs}}$$

$$\frac{}{\cdot \vdash \cdot} \tag{43}$$

$$\frac{\mathit{cdecs} \vdash \mathit{adecs}}{\mathit{cdecs}, \mathit{unitid} :_{\langle i \rangle} \mathit{intf} \vdash \mathit{adecs}, \mathit{unitid} : \mathit{intf}} \tag{44}$$

$$\boxed{\vdash \mathit{lscript}\ \textsf{ok}}$$

$$\frac{\vdash \mathit{assms}\ \textsf{ok}}{\vdash \textsf{combine}\ \mathit{assms}\ \textsf{ok}} \tag{45}$$

$$\frac{\begin{array}{c}\vdash \mathit{assms}\ \mathsf{ok} \\ \mathit{assms} = \mathit{assm}_1; \cdots; \mathit{assm}_m \\ \{\mathit{unitid}_1, \ldots, \mathit{unitid}_n\} \subset \mathrm{dom}(\mathit{assm}_1) \cup \cdots \cup \mathrm{dom}(\mathit{assm}_m)\end{array}}{\vdash \mathsf{from}\ \mathit{assms}\ \mathsf{select}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{ok}} \tag{46}$$

$\boxed{\mathit{adecs} \vdash \mathit{assms}\ \mathsf{ok}}$

$$\frac{\vdash \mathit{adecs}\ \mathsf{ok}}{\mathit{adecs} \vdash \cdot\ \mathsf{ok}} \tag{47}$$

$$\frac{\mathit{adecs} \vdash \mathit{assm}\ \mathsf{ok} \quad \vdash \mathit{assms}\ \mathsf{ok}}{\mathit{adecs} \vdash \mathit{assm}; \mathit{assms}\ \mathsf{ok}} \tag{48}$$

Rule 48: The first assembly may make reference to units declared in *adecs*. Subsequent assemblies must be well-formed in isolation.

$\boxed{\vdash \mathit{cdecs}\ \mathsf{ok}}$

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \tag{49}$$

$$\frac{\begin{array}{c}\vdash \mathit{cdecs}\ \mathsf{ok} \\ \mathit{unitids} \notin \mathrm{dom}(\mathit{cdecs}) \\ \mathit{cdecs} \vdash \mathit{adecs} \\ \mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}\end{array}}{\vdash \mathit{cdecs}, \mathit{unitid} :_{\langle i \rangle} \mathit{intf}\ \mathsf{ok}} \tag{50}$$

$\boxed{\vdash \mathit{deps}\ \mathsf{ok}}$

$$\frac{\begin{array}{c}\vdash \mathit{assm}\ \mathsf{ok} \\ \{\mathit{unitid}_1, \ldots, \mathit{unitid}_n\} \subset \mathrm{dom}(\mathit{assm})\end{array}}{\vdash \mathit{assm}; \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{ok}} \tag{51}$$

D Realization for The Typed Semantics

D.1 Realization of the IL Static Semantics for TS

Definition 5. *The domain of a top-level declarations list,* dom(*tdecs*)*, is defined by:*

Function	*Definition*		
dom(*tdecs*)	$\mathrm{dom}(\mathit{tdec}_1, \ldots, \mathit{tdec}_n)$	$=$	$\{\mathrm{dom}(\mathit{tdec}_1), \ldots, \mathrm{dom}(\mathit{tdec}_n)\}$
dom(*tdec*)	$\mathrm{dom}(\mathit{sigid} : \mathsf{Sig} = \mathit{sig})$	$=$	$\mathit{sigid}.$

$$\boxed{decs \vdash intf : \mathsf{Intf}}$$

$$\frac{var \notin \mathrm{BV}(decs) \quad decs \vdash sdecs\ \mathsf{ok} \quad decs, var : [sdecs] \vdash tdecs\ \mathsf{ok}}{decs \vdash (var : [sdecs]; tdecs) : \mathsf{Intf}} \tag{52}$$

$$\boxed{decs \vdash tdecs\ \mathsf{ok}}$$

$$\frac{\begin{array}{c}\vdash decs\ \mathsf{ok} \\ sigid_1, \ldots, sigid_n \text{ are distinct} \\ decs \vdash sig_1 : \mathsf{Sig} \quad \cdots \quad decs \vdash sig_n : \mathsf{Sig}\end{array}}{decs \vdash (sigid_1 : \mathsf{Sig} = sig_1, \ldots, sigid_n : \mathsf{Sig} = sig_n)\ \mathsf{ok}} \tag{53}$$

$$\boxed{decs \vdash impl : intf}$$

$$\frac{var \notin \mathrm{BV}(decs) \quad decs \vdash mod : [sdecs] \quad decs, var : [sdecs] \vdash tdecs\ \mathsf{ok}}{decs \vdash mod : (var : [sdecs]; tdecs)} \tag{54}$$

$$\boxed{decs \vdash intf \equiv intf' : \mathsf{Intf}}$$

$$\frac{\begin{array}{c}var \notin \mathrm{BV}(decs) \quad var' \notin \mathrm{BV}(decs) \cup \{var\} \\ decs \vdash sdecs \equiv sdecs' \quad decs, var : [sdecs] \vdash var : sig \\ decs, var : [sdecs], var' : sig \vdash tdecs \equiv tdecs'\end{array}}{decs \vdash (var : [sdecs]; tdecs) \equiv (var' : [sdecs']; tdecs') : \mathsf{Intf}} \tag{55}$$

Rule 55: The signature *sig* should be the fully selfified signature for *var*, in order to maximize type sharing when signatures in *tdecs* and $tdecs'$ are compared.

$$\boxed{decs \vdash tdecs \equiv tdecs'}$$

$$\frac{decs \vdash tdecs \supset tdecs' \quad decs \vdash tdecs' \supset tdecs}{decs \vdash tdecs \equiv tdecs'} \tag{56}$$

$$\boxed{decs \vdash intf \leq intf' : \mathsf{Intf}}$$

$$\frac{\begin{array}{c}var \notin \mathrm{BV}(decs) \quad var' \notin \mathrm{BV}(decs) \cup \{var\} \\ decs \vdash sdecs \leq sdecs' \quad decs, var : [sdecs] \vdash var : sig \\ decs, var : [sdecs], var' : sig \vdash tdecs \supset tdecs'\end{array}}{decs \vdash (var : [sdecs]; tdecs) \leq (var' : [sdecs']; tdecs') : \mathsf{Intf}} \tag{57}$$

Rule 57: The signature *sig* should be the fully selfified signature for *var*, in order to maximize type sharing when signatures in *tdecs* and $tdecs'$ are compared.

$$\boxed{decs \vdash tdecs \supset tdecs'}$$

$$\frac{decs \vdash tdecs\ \mathsf{ok}}{decs \vdash tdecs \supset \cdot} \tag{58}$$

$$\frac{\begin{array}{c} sigid \notin \mathrm{dom}(tdecs') \\ tdecs = (tdecs_1, sigid = sig' : \mathsf{Sig}, tdecs_2) \quad decs \vdash sig \equiv sig' : \mathsf{Sig} \\ decs \vdash tdecs \supset tdecs' \end{array}}{decs \vdash tdecs \supset (tdecs', sigid = sig : \mathsf{Sig})} \tag{59}$$

$$\boxed{adecs \vdash decs}$$

$$\frac{}{\cdot \vdash \cdot} \tag{60}$$

$$\frac{\begin{array}{c} adecs \vdash \Gamma \quad \overline{unitid} \notin \mathrm{BV}(\Gamma) \quad var \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash sdecs\ \mathsf{ok} \quad \Gamma, var : [sdecs] \vdash tdecs\ \mathsf{ok} \end{array}}{adecs, unitid : (var : [sdecs]; tdecs) \vdash \Gamma, \overline{unitid} : [sdecs]} \tag{61}$$

D.2 Realization of the Elaborator for TS

Definition 6. *The domain of an elaboration context,* dom(*udecs*)*, is defined by:*

Function	*Definition*		
$\mathrm{dom}(udecs)$	$\mathrm{dom}(udec_1, \ldots, udec_n)$	$=$	$\{\mathrm{dom}(udec_1), \ldots, \mathrm{dom}(udec_n)\}$
$\mathrm{dom}(udec)$	$\mathrm{dom}(sdec)$	$=$	$\mathrm{dom}(sdec)$
	$\mathrm{dom}(tdec)$	$=$	$\mathrm{dom}(tdec)$
$\mathrm{dom}(sdec)$	$\mathrm{dom}(lab \triangleright dec)$	$=$	$lab.$

Definition 7. *The set of bound variables in an elaboration context,* BV(*udecs*)*, is defined by:*

Function	*Definition*		
$\mathrm{BV}(udecs)$	$\mathrm{BV}(\cdot)$	$=$	$\emptyset$
	$\mathrm{BV}(udecs, sdec)$	$=$	$\mathrm{BV}(udecs) \cup \{\mathrm{BV}(sdec)\}$
	$\mathrm{BV}(udecs, tdec)$	$=$	$\mathrm{BV}(udecs)$
$\mathrm{BV}(sdec)$	$\mathrm{BV}(lab \triangleright dec)$	$=$	$\mathrm{BV}(dec).$

Definition 8. *The notation* $\{phrase/var\}tphrase$ *denotes the capture-free substitution of phrase for free occurrences of var within tphrase, where tphrase is defined by:*

$$\begin{array}{rcl} tphrase & ::= & sbnds \\ & & sdecs \\ & & tdecs. \end{array}$$

Definition 9. *We define the application of a renaming to a tphrase,* $\{\sigma\}tphrase$*, by:*

$$\begin{array}{rcl} \{\cdot\}tphrase & = & tphrase \\ \{\sigma, var/var'\}tphrase & = & \{\sigma\}(\{var/var'\}tphrase) \end{array}$$

The TS elaborator handles shadowing of external language identifiers using an operation of syntactic concatenation with renaming. The notation is $\mathit{sbnds}{+\!\!+}\mathit{sbnds}' : \mathit{sdecs}{+\!\!+}\mathit{sdecs}'$ and the operation renames shadowed labels so that they are unavailable to identifier lookup but so that the result of elaboration may continue to refer to "hidden" components through their variables. We can simply drop shadowed signature identifiers as *tdecs* do not bind variables.

Definition 10. *We define the shadowing operation $\mathit{tdecs}{+\!\!+}\mathit{tdecs}'$ by:*

$$
\begin{array}{l}
(\cdot{+\!\!+}\mathit{tdecs}') = \mathit{tdecs}' \\
((\mathit{sigid} : \mathsf{Sig} = \mathit{sig}, \mathit{tdecs}){+\!\!+}\mathit{tdecs}') = \\
\quad \begin{cases} \mathit{sigid} : \mathsf{Sig} = \mathit{sig}, \mathit{tdecs}'' & \text{if } \mathit{sigid} \notin \mathrm{dom}(\mathit{tdecs}'') \\ \mathit{tdecs}'' & \text{otherwise} \end{cases} \\
\quad \text{where } \mathit{tdecs}'' = \mathit{tdecs}{+\!\!+}\mathit{tdecs}'.
\end{array}
$$

$$\boxed{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \leadsto \mathit{impl} : \mathit{intf}}$$

$$
\frac{\begin{array}{c}\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids} \leadsto \mathit{udecs}, \sigma \\ \mathit{udecs} \vdash \mathit{topdec} \leadsto \mathit{sbnds} : (\mathit{sdecs}; \mathit{tdecs}) \\ \mathit{impl} = [\{\sigma\}\mathit{sbnds}] \quad \mathit{udecs} \vdash \{\sigma\}\mathit{sdecs}; \{\sigma\}\mathit{tdecs} \leadsto \mathit{intf} : \mathsf{Intf}\end{array}}{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \leadsto \mathit{impl} : \mathit{intf}} \tag{62}
$$

$$\boxed{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topspec} \leadsto \mathit{intf} : \mathsf{Intf}}$$

$$
\frac{\begin{array}{c}\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids} \leadsto \mathit{udecs}, \sigma \\ \mathit{udecs} \vdash \mathit{topspec} \leadsto \mathit{sdecs}; \mathit{tdecs} \\ \mathit{udecs} \vdash \{\sigma\}\mathit{sdecs}; \{\sigma\}\mathit{tdecs} \leadsto \mathit{intf} : \mathsf{Intf}\end{array}}{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topspec} \leadsto \mathit{intf} : \mathsf{Intf}} \tag{63}
$$

$$\boxed{\mathit{udecs} \vdash \mathit{sdecs}; \mathit{tdecs} \leadsto \mathit{intf} : \mathsf{Intf}}$$

$$
\frac{\begin{array}{c}\mathit{var} \notin \mathrm{BV}(\mathit{decs}) \\ \mathit{sdecs} = \mathit{lab}_1 \rhd \mathit{dec}_1, \ldots, \mathit{lab}_n \rhd \mathit{dec}_n \\ \mathit{var}_1 = \mathrm{BV}(\mathit{dec}_1) \quad \cdots \quad \mathit{var}_n = \mathrm{BV}(\mathit{dec}_n) \\ \mathit{tdecs}' = \{\mathit{var}.\mathit{lab}_1/\mathit{var}_1\}\cdots\{\mathit{var}.\mathit{lab}_n/\mathit{var}_n\}\mathit{tdecs}\end{array}}{\mathit{udecs} \vdash \mathit{sdecs}; \mathit{tdecs} \leadsto (\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}') : \mathsf{Intf}} \tag{64}
$$

$$\boxed{\mathit{decs} \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \leadsto \mathit{impl}}$$

$$
\frac{\begin{array}{c}\mathit{var}_0 \neq \mathit{var} \\ \mathit{decs}, \mathit{var}_0 : [\mathit{sdecs}_0] \vdash_{\mathsf{sub}} \mathit{var}_0 : [\mathit{sdecs}_0] \preceq [\mathit{sdecs}] \leadsto \mathit{mod}' : \mathit{sig} \\ \mathit{decs}, \mathit{var}_0 : [\mathit{sdecs}_0], \mathit{var} : \mathit{sig} \vdash \mathit{tdecs}_0 \supset \mathit{tdecs} \\ \mathit{mod} = (((\lambda \mathit{var}_0 : [\mathit{sdecs}_0].\mathit{mod}')\ \mathit{mod}_0) : [\mathit{sdecs}])\end{array}}{\mathit{decs} \vdash \mathit{mod}_0 : (\mathit{var}_0 : [\mathit{sdecs}_0]; \mathit{tdecs}_0) \preceq (\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}) \leadsto \mathit{mod}} \tag{65}
$$

Rule 65: The TS coercion compilation judgement produces the "leaky" signature *sig* for implementing SML transparent ascription. We use it to maximize sharing when signatures in *tdecs* and tdecs_0 are compared.

$$\boxed{udecs \vdash topdec \rightsquigarrow sbnds : (sdecs; tdecs)}$$

$$\frac{\begin{array}{c} udecs \vdash strdec \rightsquigarrow sbnds : sdecs \\ \langle udecs, sdecs \vdash topdec \rightsquigarrow sbnds' : (sdecs'; tdecs)\rangle \end{array}}{\begin{array}{l} udecs \vdash strdec\ \langle topdec\rangle \rightsquigarrow \\ \quad sbnds\langle +\!\!+ sbnds'\rangle : (sdecs\langle +\!\!+ sdecs'\rangle; \cdot\langle +\!\!+ tdecs\rangle) \end{array}} \tag{66}$$

$$\frac{\begin{array}{c} udecs \vdash sigbind \rightsquigarrow tdecs \\ \langle udecs, tdecs \vdash topdec \rightsquigarrow sbnds : (sdecs; tdecs')\rangle \end{array}}{\begin{array}{l} udecs \vdash \textsf{signature}\ sigbind\ \langle topdec\rangle \rightsquigarrow \\ \quad \cdot\langle, sbnds\rangle : (\cdot\langle, sdecs\rangle; tdecs\langle +\!\!+ tdecs'\rangle) \end{array}} \tag{67}$$

$$\frac{\begin{array}{c} udecs \vdash funbind \rightsquigarrow sbnds : sdecs \\ \langle udecs, sdecs \vdash topdec \rightsquigarrow sbnds' : (sdecs'; tdecs)\rangle \end{array}}{\begin{array}{l} udecs \vdash \textsf{functor}\ funbind\ \langle topdec\rangle \rightsquigarrow \\ \quad sbnds\langle +\!\!+ sbnds'\rangle : (sdecs\langle +\!\!+ sdecs'\rangle; \cdot\langle +\!\!+ tdecs\rangle) \end{array}} \tag{68}$$

$$\boxed{udecs \vdash topspec \rightsquigarrow sdecs; tdecs}$$

$$\frac{udecs \vdash spec \rightsquigarrow sdecs}{udecs \vdash spec \rightsquigarrow sdecs; \cdot} \tag{69}$$

$$\frac{udecs \vdash funspec \rightsquigarrow sdecs}{udecs \vdash \textsf{functor}\ funspec \rightsquigarrow sdecs; \cdot} \tag{70}$$

$$\frac{udecs \vdash sigbind \rightsquigarrow tdecs}{udecs \vdash \textsf{signature}\ sigbind \rightsquigarrow \cdot; tdecs} \tag{71}$$

$$\frac{\begin{array}{c} udecs \vdash topspec_1 \rightsquigarrow sdecs_1; tdecs_1 \\ udecs, sdecs_1, tdecs_1 \vdash topspec_2 \rightsquigarrow sdecs_2; tdecs_2 \\ udecs \vdash decs \\ decs \vdash sdecs_1, sdecs_2\ \textsf{ok} \quad decs \vdash tdecs_1, tdecs_2\ \textsf{ok} \end{array}}{udecs \vdash topspec_1\ \ topspec_2 \rightsquigarrow sdecs_1, sdecs_2; tdecs_1, tdecs_2} \tag{72}$$

Rule 72: Because of include, there is no way to restrict the syntax to ensure that the concatenation $sdecs_1, sdecs_2$ is well-formed.

$$\boxed{udecs \vdash sigbind \rightsquigarrow tdecs}$$

$$\frac{\begin{array}{c} udecs \vdash sigexp \rightsquigarrow sig : \textsf{Sig} \\ \langle udecs \vdash sigbind \rightsquigarrow tdecs \quad sigid \notin \mathrm{dom}(tdecs)\rangle \end{array}}{udecs \vdash sigid = sigexp\ \langle \textsf{and}\ sigbind\rangle \rightsquigarrow sigid = sig\langle, tdecs\rangle} \tag{73}$$

$$\boxed{\mathit{udecs} \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}$$

$$\frac{\mathit{udecs} \vdash_{\mathsf{ctx}} \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}{\mathit{udecs} \vdash \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{74}$$

Rule 74: We add this rule to the TS rules for elaborating signature expressions.

$$\boxed{\mathit{udecs} \vdash \mathit{funspec} \rightsquigarrow \mathit{sdecs}}$$

$$\frac{\begin{array}{c}\mathit{udecs} \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig} \quad \mathit{var} \notin \mathrm{BV}(\mathit{udecs}) \\ \mathit{udecs}, \overline{\mathit{strid}} \rhd \mathit{var} : \mathit{sig} \vdash \mathit{sigexp}' \rightsquigarrow \mathit{sig}' : \mathsf{Sig} \\ \langle \mathit{udecs} \vdash \mathit{funspec} \rightsquigarrow \mathit{sdecs} \quad \overline{\mathit{funid}} \notin \mathrm{dom}(\mathit{sdecs})\rangle\end{array}}{\begin{array}{l}\mathit{udecs} \vdash \mathit{funid}(\mathit{strid} : \mathit{sigexp}) : \mathit{sigexp}' \ \langle \mathsf{and}\ \mathit{funspec}\rangle \rightsquigarrow \\ \overline{\mathit{funid}} : (\mathit{var} : \mathit{sig} \rightharpoonup \mathit{sig}')\langle, \mathit{sdecs}\rangle\end{array}} \tag{75}$$

$$\boxed{\mathit{udecs} \vdash_{\mathsf{ctx}} \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}$$

$$\frac{}{\mathit{udecs}, \mathit{sigid} : \mathsf{Sig} = \mathit{sig} \vdash_{\mathsf{ctx}} \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{76}$$

$$\frac{\mathit{sigid}' \neq \mathit{sigid} \quad \mathit{udecs} \vdash_{\mathsf{ctx}} \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}{\mathit{udecs}, \mathit{sigid}' : \mathsf{Sig} = \mathit{sig}' \vdash_{\mathsf{ctx}} \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{77}$$

$$\frac{\mathit{udecs} \vdash_{\mathsf{ctx}} \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}{\mathit{udecs}, \mathit{sdec} \vdash_{\mathsf{ctx}} \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{78}$$

$$\boxed{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids} \rightsquigarrow \mathit{udecs}, \sigma}$$

$$\frac{\begin{array}{c}\mathit{unitid}_1, \ldots, \mathit{unitid}_n \in \mathrm{dom}(\mathit{adecs}) \\ \mathit{adecs} \vdash \mathit{decs} \quad \mathit{decs} = \mathit{dec}_1, \ldots, \mathit{dec}_m \\ \mathit{udecs}_0 = 1 \rhd \mathit{dec}_1, \ldots, 1 \rhd \mathit{dec}_m \\ \mathit{adecs} = \mathit{adecs}_1', \mathit{unitid}_1 : (\mathit{var}_1 : [\mathit{sdecs}_1]; \mathit{tdecs}_1), \mathit{adecs}_1'' \\ \mathit{decs} \vdash \overline{\mathit{unitid}_1} : \mathit{sig}_1 \quad \mathit{udecs}_1 = 1^\star \rhd \mathit{var}_1 : \mathit{sig}_1, \mathit{tdecs}_1 \\ \vdots \\ \mathit{adecs} = \mathit{adecs}_n', \mathit{unitid}_n : (\mathit{var}_n : [\mathit{sdecs}_n]; \mathit{tdecs}_n), \mathit{adecs}_n'' \\ \mathit{decs} \vdash \overline{\mathit{unitid}_n} : \mathit{sig}_n \quad \mathit{udecs}_n = 1^\star \rhd \mathit{var}_n : \mathit{sig}_n, \mathit{tdecs}_n \\ \mathit{udecs} = \mathit{udecs}_0, \mathit{udecs}_1, \ldots, \mathit{udecs}_n \\ \sigma = \overline{\mathit{unitid}_1}/\mathit{var}_1, \ldots, \overline{\mathit{unitid}_n}/\mathit{var}_n\end{array}}{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \rightsquigarrow \mathit{udecs}, \sigma} \tag{79}$$

Rule 79: The signature sig_i should the fully selfified signature for $\overline{\mathit{unitid}_i}$, in order to ensure that types projected from the "open" modules var_i are equivalent to the corresponding types in unitid_i.

$$\boxed{\vdash \mathit{udecs}\ \mathsf{ok}}$$

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \tag{80}$$

$$\frac{\vdash \mathit{udecs}\ \mathsf{ok} \quad \mathit{udecs} \vdash \mathit{decs} \quad \mathit{decs} \vdash \mathit{dec}\ \mathsf{ok}}{\vdash \mathit{udecs}, \mathit{lab} \triangleright \mathit{dec}\ \mathsf{ok}} \tag{81}$$

$$\frac{\vdash \mathit{udecs}\ \mathsf{ok} \quad \mathit{udecs} \vdash \mathit{decs} \quad \mathit{decs} \vdash \mathit{tdec}\ \mathsf{ok}}{\vdash \mathit{udecs}, \mathit{tdec}\ \mathsf{ok}} \tag{82}$$

$$\boxed{\mathit{udecs} \vdash \mathit{decs}}$$

$$\frac{\mathit{udecs} \vdash \mathit{decs}}{\mathit{udecs}, \mathit{lab} \triangleright \mathit{dec} \vdash \mathit{decs}, \mathit{dec}} \tag{83}$$

$$\frac{\mathit{udecs} \vdash \mathit{decs}}{\mathit{udecs}, \mathit{tdec} \vdash \mathit{decs}} \tag{84}$$

D.3 Realization of the Linker for TS

Definition 11. *The structure mod_{basis} must satisfy $\vdash mod_{basis} : sig_{basis}$; in particular, it must contain at least the following fields:*

$$\begin{aligned}[\overline{\mathsf{Bind}}^\star &= [\mathsf{tag} \triangleright var = \mathsf{new\_tag}[\mathsf{Unit}], \overline{\mathsf{Bind}} = \mathsf{tag}(var, \{\})],\\ \overline{\mathsf{Match}}^\star &= [\mathsf{tag} \triangleright var = \mathsf{new\_tag}[\mathsf{Unit}], \overline{\mathsf{Match}} = \mathsf{tag}(var, \{\})],\\ \mathsf{fail}^\star &= [\mathsf{tag} \triangleright var = \mathsf{new\_tag}[\mathsf{Unit}], \mathsf{fail} = \mathsf{tag}(var, \{\})]].\end{aligned}$$

$$\boxed{\vdash \mathit{assm} \rightsquigarrow \mathit{prog}}$$

$$\frac{\vdash \mathit{assm} \rightsquigarrow \mathit{bnd}_1, \ldots, \mathit{bnd}_n : \mathit{decs} \quad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\vdash \mathit{assm} \rightsquigarrow [1 \triangleright \mathit{bnd}_1, \ldots, n \triangleright \mathit{bnd}_n, \mathsf{it} \triangleright \mathit{var} = \{\}].\mathsf{it} : \{\}} \tag{85}$$

$$\boxed{\vdash \mathit{assm} \rightsquigarrow \mathit{bnds} : \mathit{decs}}$$

$$\frac{}{\vdash \cdot \rightsquigarrow \cdot : \cdot} \tag{86}$$

$$\frac{\vdash \mathit{assm} \rightsquigarrow \mathit{bnds} : \mathit{decs}}{\begin{array}{l}\vdash \mathit{assm}, \mathsf{basis} : (\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}) \rightsquigarrow \\ \quad (\mathit{bnds}, \overline{\mathsf{basis}} = mod_{basis}) : (\mathit{decs}, \overline{\mathsf{basis}} : [\mathit{sdecs}])\end{array}} \tag{87}$$

Rule 87: Since $\vdash \mathit{assm}$ complete, $[\mathit{sdecs}]$ is equivalent to sig_{basis}.

$$\frac{\begin{array}{c} \mathit{unitid} \neq \mathsf{basis} \\ \vdash \mathit{assm} \rightsquigarrow \mathit{bnds} : \mathit{decs} \\ \mathit{intf} = \mathit{var} : [\mathit{sdecs}]; \mathit{tdecs} \\ \mathit{unite} = \langle\mathsf{internal}\rangle\ \mathsf{require}\ \mathit{unitids}\ \mathsf{in}\ \mathit{mod} \end{array}}{\begin{array}{l} \vdash \mathit{assm}, \mathit{unitid} : \mathit{intf} = \mathit{unite} \rightsquigarrow \\ \quad (\mathit{bnds}, \overline{\mathit{unitid}} = \mathit{mod}) : (\mathit{decs}, \overline{\mathit{unitid}} : [\mathit{sdecs}]) \end{array}} \tag{88}$$

$\boxed{\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}}$

$$\frac{\overline{\mathit{unitid}} \in \mathrm{FV}(\mathit{intf})}{\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}} \tag{89}$$

$\boxed{\vdash \mathit{impl}\ \mathsf{requires}\ \mathit{unitid}}$

$$\frac{\overline{\mathit{unitid}} \in \mathrm{FV}(\mathit{mod})}{\vdash \mathit{mod}\ \mathsf{requires}\ \mathit{unitid}} \tag{90}$$

$\boxed{\vdash \mathit{prog}\ \mathsf{ok}}$

$$\frac{\vdash \mathit{exp} : \{\}}{\vdash \mathit{exp} : \{\}\ \mathsf{ok}} \tag{91}$$

E Realization for The Definition

E.1 Realization of the IL Static Semantics for TD

We adopt the following notation:

- We write $(\cdot \text{ of } \cdot)$ for projection from TDIL objects; for example, T of B means "the type names component of B".
- The notation tynames A denotes the set of free type names in A. We adopt the TD notation $\varphi(A)$ for the application of a TD realization $\varphi : \text{TyName} \to \text{TypeFcn}$ to a semantic object A.
- We adopt the TD notations $A + A'$ for the modification of one map by another and $A \oplus A'$ for modification that also extends T of A to include the type names of A'.
- We adopt the TD notation $E(\cdot)$ for long identifier lookup and define the function $UE : \text{Path} \rightharpoonup \text{TyName}$ for path lookup by:

$$\begin{array}{l} UE(\mathit{unitid}.\mathit{longtycon}) = t \\ \quad \text{if } UE(\mathit{unitid}) = (T, F, G, E), L \\ \quad \text{and } E(\mathit{longtycon}) = (t, VE) \\ UE(\mathit{unitid}.n) = t \\ \quad \text{if } UE(\mathit{unitid}) = (B, L) \\ \quad \text{and } L(n) = t. \end{array}$$

- In addition to projection, we write ($\cdot$ of UE) for the sets of internal and external names bound by UE:

$$\begin{array}{rcl} T \text{ of } UE &=& \bigcup\{T \text{ of } B \ ; \ (B,L) \in \mathrm{rng}(UE)\} \\ \mathit{paths} \text{ of } UE &=& \{\mathit{path} \ ; \ UE(\mathit{path}) = t\}. \end{array}$$

$$\boxed{\Gamma \vdash \mathit{intf} : \mathsf{Intf}}$$

$$\frac{\begin{array}{c} \vdash \Gamma \ \mathsf{ok} \\ \mathrm{rng}(IP) \subset \mathit{paths} \text{ of } \Gamma \\ \vdash B \ \mathsf{ok} \quad \text{tyvars } B = \emptyset \quad \text{tynames } B \subset \mathrm{dom}(IP) \\ \text{tynames } L \subset T \text{ of } B \end{array}}{\Gamma \vdash (IP, B, L) : \mathsf{Intf}} \tag{92}$$

$$\boxed{\Gamma \vdash \mathit{impl} : \mathit{intf}}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathsf{open} \ \mathit{unitids} \Rightarrow B \\ B \vdash \mathit{topdec} \Rightarrow B' \\ \Gamma \vdash B' \Rightarrow \mathit{intf} : \mathsf{Intf} \end{array}}{\Gamma \vdash \mathsf{open} \ \mathit{unitids} \ \mathsf{in} \ \mathit{topdec} : \mathit{intf}} \tag{93}$$

$$\frac{\Gamma \vdash \mathit{impl} : \mathit{intf}' \quad \Gamma \vdash \mathit{intf}' \leq \mathit{intf} : \mathsf{Intf}}{\Gamma \vdash (\mathit{impl} : \mathit{intf}') : \mathit{intf}} \tag{94}$$

$$\boxed{\Gamma \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}}$$

$$\frac{\Gamma \vdash \mathit{intf} \Rightarrow B, L \quad \Gamma \vdash \mathit{intf}' \Rightarrow B, L}{\Gamma \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}} \tag{95}$$

$$\boxed{\Gamma \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathit{intf} \Rightarrow B, L \quad B = T, F, G, E \\ \mathit{unitid} \notin \mathrm{dom}(\Gamma) \quad \Gamma + \{\mathit{unitid} \mapsto (B, L)\} \vdash \mathit{intf}' \Rightarrow (T', F', G', E'), L' \\ \mathrm{dom}(F) \supset \mathrm{dom}(F') \quad \forall \mathit{funid} \in \mathrm{dom}(F').F'(\mathit{funid}) \geq F(\mathit{funid}) \\ \mathrm{dom}(G) \supset \mathrm{dom}(G') \quad \forall \mathit{sigid} \in \mathrm{dom}(G').G'(\mathit{sigid}) \geq G(\mathit{sigid}) \\ (T')E' \geq E'' \text{ using } \varphi \quad E'' \prec E \\ \mathrm{dom}(L) \supset \mathrm{dom}(L') \quad \forall n \in \mathrm{dom}(L').\, \varphi(L'(n)) = L(n) \end{array}}{\Gamma \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}} \tag{96}$$

Rule 96: The context is extended to ensure $T \cap T' = \emptyset$ without disturbing sharing between T and L or T' and L'.

$$\boxed{adecs \vdash \Gamma}$$

$$\frac{}{\cdot \vdash \{\}} \tag{97}$$

$$\frac{adecs \vdash \Gamma \quad unitid \notin \mathrm{dom}(\Gamma) \quad \Gamma \vdash intf \Rightarrow B, L}{adecs, unitid : intf \vdash \Gamma + \{unitid \mapsto (B, L)\}} \tag{98}$$

$$\boxed{\Gamma \vdash B \Rightarrow intf : \mathsf{Intf}}$$

$$\frac{\begin{array}{c} \vdash \Gamma\ \mathsf{ok} \quad \vdash B\ \mathsf{ok} \quad \mathrm{tyvars}\ B = \emptyset \\ \mathrm{dom}(IP) = \mathrm{tynames}\ B \subset T \text{ of } \Gamma \\ \forall t \in \mathrm{dom}(IP).\Gamma(IP(t)) = t \\ B = T, F, G, E \quad T' = \{t \in T\ ;\ \exists longtycon.E(longtycon) = (t, VE)\} \\ \mathrm{rng}(L) = T \setminus T' \end{array}}{\Gamma \vdash B \Rightarrow IP, B, L} \tag{99}$$

Rule 99: Assemblies containing inferred interfaces may be elaborated but not linked, so any choice for $\mathrm{dom}(L)$ will do. Interface equivalence is defined in terms of internal names rather than external names, so any choice of $\mathrm{rng}(IP)$ will do.

$$\boxed{\Gamma \vdash intf \Rightarrow B, L}$$

$$\frac{\begin{array}{c} \Gamma \vdash (IP, B, L) : \mathsf{Intf} \\ \mathrm{dom}(IP) \cap (T \text{ of } \Gamma) = (T \text{ of } B) \cap (T \text{ of } \Gamma) = \emptyset \\ \varphi(t) = \begin{cases} \Gamma(IP(t)) & \text{if } t \in \mathrm{dom}(IP) \\ t & \text{otherwise} \end{cases} \end{array}}{\Gamma \vdash IP, B, L \Rightarrow \varphi(B), L} \tag{100}$$

Rule 100: The side condition $\mathrm{dom}(IP) \cap (T \text{ of } \Gamma) = (T \text{ of } B) \cap (T \text{ of } \Gamma) = \emptyset$ can always be satisfied by renaming bound type names.

$$\boxed{\Gamma \vdash \mathsf{open}\ unitids \Rightarrow B}$$

$$\frac{\begin{array}{c} \vdash \Gamma\ \mathsf{ok} \quad B_0 = (T \text{ of } \Gamma), \{\}, \{\}, \{\} \\ B_1 = B \text{ of } (\Gamma(unitid_1)) \\ \vdots \\ B_n = B \text{ of } (\Gamma(unitid_n)) \end{array}}{\Gamma \vdash \mathsf{open}\ unitid_1 \cdots unitid_n \Rightarrow B_0 + B_1 + \cdots + B_n} \tag{101}$$

$$\boxed{\vdash \Gamma\ \mathsf{ok}}$$

$$\frac{\begin{array}{l}\forall \mathit{unitid} \mapsto (B, L) \in \Gamma. \\ \quad \vdash B\ \mathsf{ok}, \quad \text{tyvars } B = \emptyset, \\ \quad \text{tynames } B \subset (T \text{ of } \Gamma), \text{ and} \\ \quad \text{tynames } L \subset (T \text{ of } B) \\ \forall \mathit{unitid}, \mathit{unitid}' \in \text{dom}(\Gamma). \\ \quad \text{If } \mathit{unitid} \neq \mathit{unitid}', \\ \quad \text{then } (T \text{ of } B \text{ of } \Gamma(\mathit{unitid})) \cap (T \text{ of } B \text{ of } \Gamma(\mathit{unitid}')) = \emptyset\end{array}}{\vdash \Gamma\ \mathsf{ok}} \tag{102}$$

E.2 Realization of the Elaborator for TD

$$\boxed{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \rightsquigarrow \mathit{impl} : \mathit{intf}}$$

$$\frac{\begin{array}{c}\mathit{impl} = \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \\ \mathit{adecs} \vdash \Gamma \quad \Gamma \vdash \mathit{impl} : \mathit{intf}\end{array}}{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \rightsquigarrow \mathit{impl} : \mathit{intf}} \tag{103}$$

Rule 103: A compiled unit contains source code that is evaluated after completion.

$$\boxed{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf}}$$

$$\frac{\begin{array}{c}\mathit{adecs} \vdash \Gamma \quad \Gamma \vdash \mathsf{open}\ \mathit{unitids} \Rightarrow B \\ B \vdash \mathit{topspec} \Rightarrow B' \quad \Gamma \vdash B' \Rightarrow \mathit{intf} : \mathsf{Intf}\end{array}}{\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf}} \tag{104}$$

$$\boxed{B \vdash \mathit{topspec} \Rightarrow B'}$$

$$\frac{B \vdash \mathit{spec} \Rightarrow E \quad B' = T \text{ of } E, \{\}, \{\}, E \quad \text{tyvars } B' = \emptyset}{B \vdash \mathit{spec} \Rightarrow B'} \tag{105}$$

$$\frac{B \vdash \mathit{funspec} \Rightarrow F \quad B' = T \text{ of } F, F, \{\}, \{\} \quad \text{tyvars } B' = \emptyset}{B \vdash \mathsf{functor}\ \mathit{funspec} \Rightarrow B'} \tag{106}$$

$$\frac{B \vdash \mathit{sigbind} \Rightarrow G \quad B' = T \text{ of } G, \{\}, G, \{\} \quad \text{tyvars } B' = \emptyset}{B \vdash \mathsf{signature}\ \mathit{sigbind} \Rightarrow B'} \tag{107}$$

$$\frac{\begin{array}{c}B \vdash \mathit{topspec}_1 \Rightarrow B_1 \quad B \oplus B_1 \vdash \mathit{topspec}_2 \Rightarrow B_2 \\ \text{dom}(F \text{ of } B_1) \cap \text{dom}(F \text{ of } B_2) = \emptyset \\ \text{dom}(G \text{ of } B_1) \cap \text{dom}(G \text{ of } B_2) = \emptyset \\ \text{dom}(E \text{ of } B_1) \cap \text{dom}(E \text{ of } B_2) = \emptyset\end{array}}{B \vdash \mathit{topspec}_1\ \mathit{topspec}_2 \Rightarrow B_1 + B_2} \tag{108}$$

$$\boxed{B \vdash \mathit{funspec} \Rightarrow F}$$

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow (T)E \quad B \oplus \{\mathit{strid} \mapsto E\} \vdash \mathit{sigexp}' \Rightarrow (T')E' \\ \langle B \vdash \mathit{funspec} \Rightarrow F \quad \mathit{funid} \notin \mathrm{dom}(F) \rangle \end{array}}{\begin{array}{l} B \vdash \mathit{funid}(\mathit{strid} : \mathit{sigexp}) : \mathit{sigexp}' \ \langle \mathsf{and}\ \mathit{funspec} \rangle \Rightarrow \\ \quad \{\mathit{funid} \mapsto (T)(E, (T')E')\} \ \langle + F \rangle \end{array}} \tag{109}$$

$$\boxed{\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{impl}}$$

$$\frac{\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \quad \Gamma \vdash \mathit{intf}_0 \leq \mathit{intf} : \mathsf{Intf}}{\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow (\mathit{impl}_0 : \mathit{intf})} \tag{110}$$

E.3 Realization of the Linker for TD

$$\boxed{\vdash \mathit{assm} \rightsquigarrow \mathit{prog}}$$

$$\frac{}{\vdash \mathit{assm} \rightsquigarrow \mathit{assm}} \tag{111}$$

Rule 111: A compiled assembly contains source code that is evaluated using the rules in Appendix E.4.

$$\boxed{\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}}$$

$$\frac{\mathit{IP}(t) = \mathit{unitid}.\mathit{longtycon}}{\vdash (\mathit{IP}, B, L)\ \mathsf{requires}\ \mathit{unitid}} \tag{112}$$

$$\boxed{\vdash \mathit{impl}\ \mathsf{requires}\ \mathit{unitid}}$$

$$\frac{\mathit{unitid} \in \{\mathit{unitid}_1, \ldots, \mathit{unitid}_n\}}{\vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topdec}\ \mathsf{requires}\ \mathit{unitid}} \tag{113}$$

$$\frac{\vdash \mathit{impl}\ \mathsf{requires}\ \mathit{unitid}}{\vdash \mathit{impl} : \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}} \tag{114}$$

$$\frac{\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}}{\vdash \mathit{impl} : \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}} \tag{115}$$

$$\boxed{\vdash \mathit{prog}\ \mathsf{ok}}$$

$$\frac{\begin{array}{c} \mathit{prog} = \mathit{assm} \\ \vdash \mathit{assm}\ \mathsf{complete} \end{array}}{\vdash \mathit{prog}\ \mathsf{ok}} \tag{116}$$

E.4 Dynamic Semantic of Programs for TD

Definition 12. *The creation of dynamic TDIL "interfaces" from static TDIL objects,* $\mathrm{inter}(\cdot)$*, is defined by:*

$$\begin{array}{l}
\mathrm{inter} : \mathrm{SigEnv}_{\mathrm{STAT}} \to \mathrm{SigEnv} \\
\mathrm{inter}(G) = \{ \mathit{sigid} \mapsto \mathrm{inter}(\Sigma) \ ; \ G(\mathit{sigid}) = \Sigma \} \\
\\
\mathrm{inter} : \mathrm{Sig}_{\mathrm{STAT}} \to \mathrm{Int} \\
\mathrm{inter}((T)E) = \mathrm{inter}(E) \\
\\
\mathrm{inter} : \mathrm{Env}_{\mathrm{STAT}} \to \mathrm{Int} \\
\mathrm{inter}(\mathit{SE}, \mathit{TE}, \mathit{VE}) = \mathrm{inter}(\mathit{SE}), \mathrm{inter}(\mathit{TE}), \mathrm{inter}(\mathit{VE}) \\
\\
\mathrm{inter} : \mathrm{StrEnv}_{\mathrm{STAT}} \to \mathrm{StrInt} \\
\mathrm{inter}(\mathit{SE}) = \{ \mathit{strid} \mapsto \mathrm{inter}(E) \ ; \ \mathit{SE}(\mathit{strid}) = E \} \\
\\
\mathrm{inter} : \mathrm{TyEnv}_{\mathrm{STAT}} \to \mathrm{TyInt} \\
\mathrm{inter}(\mathit{TE}) = \{ \mathit{tycon} \mapsto \mathrm{inter}(\mathit{VE}) \ ; \ \mathit{TE}(\mathit{tycon}) = (\theta, \mathit{VE}) \} \\
\\
\mathrm{inter} : \mathrm{ValEnv}_{\mathrm{STAT}} \to \mathrm{ValInt} \\
\mathrm{inter}(\mathit{VE}) = \{ \mathit{vid} \mapsto \mathit{is} \ ; \ \mathit{VE}(\mathit{vid}) = (\sigma, \mathit{is}) \}.
\end{array}$$

Definition 13. *The thinning of a basis by a compiled interface,* $B \downarrow \mathit{intf}$*, is defined by:*

$$\begin{array}{l}
\downarrow : \mathrm{Basis} \times (\mathrm{Imports} \times \mathrm{Basis}_{\mathrm{STAT}} \times \mathrm{Labels}) \to \mathrm{Basis} \\
(F, G, E) \downarrow (\mathit{IP}, (T', F', G', E'), L) = (F \downarrow F', \mathrm{inter}(G'), E \downarrow \mathrm{inter}(E')) \\
\\
\downarrow : \mathrm{FunEnv} \times \mathrm{FunEnv}_{\mathrm{STAT}} \to \mathrm{FunEnv} \\
F \downarrow F' = \{ \mathit{funid} \mapsto F(\mathit{funid}) \ ; \ \mathit{funid} \in \mathrm{dom}(F) \cap \mathrm{dom}(F') \}
\end{array}$$

where $\downarrow : \mathrm{Env} \times \mathrm{Int} \to \mathrm{Env}$ *is defined in TD.*

$$\boxed{\vdash \mathit{prog} \Rightarrow \mathit{UE}/p, s}$$

$$\frac{(\{\}, \{\}), \{\} \vdash \mathit{assm} \Rightarrow \mathit{UE}/p, s}{\vdash \mathit{assm} \Rightarrow \mathit{UE}/p, s} \tag{117}$$

$$\boxed{s, \mathit{UE} \vdash \mathit{assm} \Rightarrow \mathit{UE}'/p, s'}$$

$$\frac{}{s, \mathit{UE} \vdash \cdot \Rightarrow \mathit{UE}, s} \tag{118}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\mathit{mem} \text{ of } s) \cap \mathrm{dom}(\mathit{mem} \text{ of } s_0) = \emptyset \\ (\mathit{ens} \text{ of } s) \cap (\mathit{ens} \text{ of } s_0) = \emptyset \\ s + s_0, \mathit{UE} + \{\mathsf{basis} \mapsto B_0\} \vdash \mathit{assm} \Rightarrow \mathit{UE}', s'\end{array}}{s, \mathit{UE} \vdash \mathsf{basis} : \mathit{intf}, \mathit{assm} \Rightarrow \mathit{UE}', s'} \tag{119}$$

Rule 119: The side conditions can always be satisfied by changing addresses and exception names in B_0.

$$\frac{\begin{array}{c}\mathrm{dom}(\mathit{mem} \text{ of } s) \cap \mathrm{dom}(\mathit{mem} \text{ of } s_0) = \emptyset \\ (\mathit{ens} \text{ of } s) \cap (\mathit{ens} \text{ of } s_0) = \emptyset \\ s + s_0, \mathit{UE} + \{\mathsf{basis} \mapsto B_0\} \vdash \mathit{assm} \Rightarrow p, s'\end{array}}{s, \mathit{UE} \vdash \mathsf{basis} : \mathit{intf}, \mathit{assm} \Rightarrow p, s'} \tag{120}$$

$$\frac{\begin{array}{c}\mathit{unitid} \neq \mathsf{basis} \\ \mathit{unite} = \langle\mathsf{internal}\rangle\ \mathsf{require}\ \mathit{unitids}\ \mathsf{in}\ \mathit{impl} \\ s, \mathit{UE} \vdash \mathit{impl} \Rightarrow B, s' \quad s', \mathit{UE} + \{\mathit{unitid} \mapsto B\} \vdash \mathit{assm} \Rightarrow \mathit{UE}', s''\end{array}}{s, \mathit{UE} \vdash \mathit{unitid} : \mathit{intf} = \mathit{unite}, \mathit{assm} \Rightarrow \mathit{UE}', s''} \tag{121}$$

$$\frac{\begin{array}{c}\mathit{unitid} \neq \mathsf{basis} \\ \mathit{unite} = \langle\mathsf{internal}\rangle\ \mathsf{require}\ \mathit{unitids}\ \mathsf{in}\ \mathit{impl} \\ s, \mathit{UE} \vdash \mathit{impl} \Rightarrow p, s'\end{array}}{s, \mathit{UE} \vdash \mathit{unitid} : \mathit{intf} = \mathit{unite}, \mathit{assm} \Rightarrow p, s'} \tag{122}$$

$$\frac{\begin{array}{c}\mathit{unitid} \neq \mathsf{basis} \\ \mathit{unite} = \langle\mathsf{internal}\rangle\ \mathsf{require}\ \mathit{unitids}\ \mathsf{in}\ \mathit{impl} \\ s, \mathit{UE} \vdash \mathit{impl} \Rightarrow B, s' \quad s', \mathit{UE} + \{\mathit{unitid} \mapsto B\} \vdash \mathit{assm} \Rightarrow p, s''\end{array}}{s, \mathit{UE} \vdash \mathit{unitid} : \mathit{intf} = \mathit{unite}, \mathit{assm} \Rightarrow p, s''} \tag{123}$$

$$\boxed{s, \mathit{UE} \vdash \mathit{impl} \Rightarrow B/p, s'}$$

$$\frac{\begin{array}{c}\mathit{UE} \vdash \mathsf{open}\ \mathit{unitids} \Rightarrow B \\ s, B \vdash \mathit{topdec} \Rightarrow B', s'\end{array}}{s, \mathit{UE} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \Rightarrow B', s'} \tag{124}$$

$$\frac{\begin{array}{c}\mathit{UE} \vdash \mathsf{open}\ \mathit{unitids} \Rightarrow B \\ s, B \vdash \mathit{topdec} \Rightarrow p, s'\end{array}}{s, \mathit{UE} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \Rightarrow p, s'} \tag{125}$$

$$\frac{s, \mathit{UE} \vdash \mathit{impl} \Rightarrow B, s'}{s, \mathit{UE} \vdash \mathit{impl} : \mathit{intf} \Rightarrow B \downarrow \mathit{intf}, s'} \tag{126}$$

$$\frac{s, \mathit{UE} \vdash \mathit{impl} \Rightarrow p, s'}{s, \mathit{UE} \vdash \mathit{impl} : \mathit{intf} \Rightarrow p, s'} \tag{127}$$

$$\boxed{\mathit{UE} \vdash \mathsf{open}\ \mathit{unitids} \Rightarrow B}$$

$$\frac{B_1 = \mathit{UE}(\mathit{unitid}_1) \quad \cdots \quad B_n = \mathit{UE}(\mathit{unitid}_n)}{\mathit{UE} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \Rightarrow B_1 + \cdots + B_n} \tag{128}$$

F Properties of the Semantics

In this Appendix, we outline a meta-theory for the semantics for separate compilation. We argue that the semantics is sound provided its stubs satisfy certain properties—are *suitable*—and that the realizations for TD and TS are suitable. Most of the meta-theory is conjecture; we leave its refinement and proof for future work.

F.1 Suitability and Soundness

Definition 14 (IL Stubs Suitability)**.** *We say that the IL stubs are suitable if:*

1. *If* $\Gamma \vdash \mathit{intf} : \mathsf{Intf}$*, then* $\vdash \Gamma\ \mathsf{ok}$*.*
2. *If* $\Gamma \vdash \mathit{impl} : \mathit{intf}$*, then* $\Gamma \vdash \mathit{intf} : \mathsf{Intf}$*.*
3. *If* $\Gamma \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}$*, then* $\Gamma \vdash \mathit{intf} : \mathsf{Intf}$ *and* $\Gamma \vdash \mathit{intf}' : \mathsf{Intf}$*.*
4. *If* $\Gamma \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}$*, then* $\Gamma \vdash \mathit{intf} : \mathsf{Intf}$ *and* $\Gamma \vdash \mathit{intf}' : \mathsf{Intf}$*.*
5. *If* $\mathit{adecs} \vdash \Gamma$*, then* $\vdash \mathit{adecs}\ \mathsf{ok}$ *and* $\vdash \Gamma\ \mathsf{ok}$*.*

Conjecture 15 (IL Soundness)**.** *If the IL stubs are suitable, then:*

1. *If* $\mathit{adecs} \vdash \mathit{assm}\ \mathsf{ok}$*, then* $\vdash \mathit{adecs}\ \mathsf{ok}$*.*
2. *If* $\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$*, then* $\vdash \mathit{adecs}\ \mathsf{ok}$*.*
3. *If* $\mathit{adecs} \vdash \mathit{unite} : \mathit{intf}$*, then* $\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$*.*
4. *If* $\mathit{adecs} \vdash \mathit{impl} : \mathit{intf}$*, then* $\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$*.*
5. *If* $\mathit{adecs} \vdash \mathit{intf} \equiv \mathit{intf}'$*, then* $\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$ *and* $\mathit{adecs} \vdash \mathit{intf}' : \mathsf{Intf}$*.*
6. *If* $\mathit{adecs} \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}$*, then* $\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$ *and* $\mathit{adecs} \vdash \mathit{intf}' : \mathsf{Intf}$*.*

Definition 16 (Elaborator Stubs Suitability)**.** *We say that the elaborator stubs are suitable if:*

1. $\vdash \mathit{intf}_{\mathit{basis}} : \mathsf{Intf}$*.*
2. *If* $\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \rightsquigarrow \mathit{impl} : \mathit{intf}$ *and* $\vdash \mathit{adecs}\ \mathsf{ok}$*, then* $\mathit{adecs} \vdash \mathit{impl} : \mathit{intf}$*.*
3. *If* $\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf}$ *and* $\vdash \mathit{adecs}\ \mathsf{ok}$*, then* $\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$*.*
4. *If* $\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{impl}$*, then* $\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0$*,* $\Gamma \vdash \mathit{intf}_0 \leq \mathit{intf} : \mathsf{Intf}$*, and* $\Gamma \vdash \mathit{impl} : \mathit{intf}$*.*

Conjecture 17 (Elaborator Soundness)**.** *If the IL and elaborator stubs are suitable, then:*

1. *If* $\vdash \mathit{assembly} \rightsquigarrow \mathit{assm}; \mathit{edecs}$*, then* $\cdot \vdash \mathit{assm}\ \mathsf{ok}$ *and* $\vdash \mathit{edecs}\ \mathsf{ok}$*.*
2. *If* $\vdash \mathit{edecs}\ \mathsf{ok}$*, then* $\mathit{edecs} \vdash \mathit{adecs}$*,* $\vdash \mathit{adecs}\ \mathsf{ok}$*, and*
   (a) *If* $\mathit{edecs} \vdash \mathit{unitexp} \rightsquigarrow \mathit{unite} : \mathit{intf}$*, then* $\mathit{adecs} \vdash \mathit{unite} : \mathit{intf}$*.*
   (b) *If* $\mathit{edecs} \vdash \mathit{intexp} \rightsquigarrow \mathit{intf} : \mathsf{Intf}$*, then* $\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$*.*

(c) *If* $edecs \vdash unite_0 : intf_0 \preceq intf \rightsquigarrow unite$*, then* $adecs \vdash unite_0 : intf_0$*,* $adecs \vdash intf_0 \leq intf : \mathsf{Intf}$*, and* $adecs \vdash unite : intf$*.*

In addition to well-formedness, suitable linking stubs have to ensure that a complete assembly can be made into an executable.

Definition 18 (Linker Stubs Suitability)**.** *We say that the linker stubs are suitable if:*

1. *If* $\vdash assm$ $\mathsf{complete}$*, then there exists a program prog such that* $\vdash assm \rightsquigarrow prog$*.*
2. *If* $\vdash intf$ $\mathsf{requires}$ $unitid$ *and* $adecs \vdash intf : \mathsf{Intf}$*, then* $unitid \in \mathrm{dom}(adecs)$*.*
3. *If* $\vdash impl$ $\mathsf{requires}$ $unitid$ *and* $adecs \vdash impl : intf$*, then* $unitid \in \mathrm{dom}(adecs)$*.*
4. *If* $\vdash assm \rightsquigarrow prog$ *and* $\vdash assm$ $\mathsf{complete}$*, then* $\vdash prog$ ok*.*

Conjecture 19 (Linker Soundness)**.** *If the IL and linker stubs are suitable, then:*

1. *If* $\vdash lscript \rightsquigarrow assm$ *and* $\vdash lscript$ ok*, then* $\vdash assm$ ok*.*
2. *If* $cdecs \vdash assms \rightsquigarrow assm$*;* $cdecs \vdash adecs$*; and* $adecs \vdash assms$ ok*, then* $adecs \vdash assm$ ok*.*
3. *If* $adecs \vdash_{deps} assm \rightsquigarrow assm'$ *and* $\vdash deps$ ok *where* $deps = (assm'', assm); unitids$ *and* $adecs = U(assm'')$*, then* $adecs \vdash assm'$ ok *and for every* $unitid \in \mathrm{dom}(assm')$*,* $\vdash deps$ $\mathsf{requires}$ $unitid$*.*
4. *If* $\vdash deps$ $\mathsf{requires}$ $unitid$ *and* $\vdash deps$ ok *where* $deps = assm; unitids$*, then* $unitid \in \mathrm{dom}(assm)$*.*
5. *If* $adecs \vdash assm$ $\mathsf{complete}$*, then* $adecs \vdash assm$ ok*.*
6. *If* $cdecs \vdash adecs$ *and* $\vdash cdecs$ ok*, then* $\vdash adecs$ ok*.*
7. *If* $adecs \vdash assms$ ok*, then* $\vdash adecs$ ok*.*

F.2 Suitability of the TS Realization

Conjecture 20. *The realization of the IL static semantics for TS is suitable:*

1. *If* $decs \vdash intf : \mathsf{Intf}$*, then* $\vdash decs$ ok*.*
2. *If* $decs \vdash tdecs$ ok*, then* $\vdash decs$ ok*.*
3. *If* $decs \vdash impl : intf$*, then* $decs \vdash intf : \mathsf{Intf}$*.*
4. *If* $decs \vdash intf \equiv intf' : \mathsf{Intf}$*, then* $decs \vdash intf : \mathsf{Intf}$ *and* $decs \vdash intf' : \mathsf{Intf}$*.*
5. *If* $decs \vdash tdecs \equiv tdecs'$*, then* $decs \vdash tdecs$ ok *and* $decs \vdash tdecs'$ ok*.*
6. *If* $decs \vdash intf \leq intf' : \mathsf{Intf}$*, then* $decs \vdash intf : \mathsf{Intf}$ *and* $decs \vdash intf' : \mathsf{Intf}$*.*
7. *If* $decs \vdash tdecs \supset tdecs'$*, then* $decs \vdash tdecs$ ok *and* $decs \vdash tdecs'$ ok*.*
8. *If* $adecs \vdash decs$*, then* $\vdash adecs$ ok *and* $\vdash decs$ ok*.*

Conjecture 21. *The realization of the elaborator for TS is suitable:*

1. $\vdash intf_{basis} : \mathsf{Intf}$*.*

2. *If* $adecs \vdash \mathsf{open}\ unitids\ \mathsf{in}\ topdec \rightsquigarrow impl : intf$ *and* $\vdash adecs\ \mathsf{ok}$, *then* $adecs \vdash impl : intf$.

3. *If* $adecs \vdash \mathsf{open}\ unitids\ \mathsf{in}\ topspec \rightsquigarrow intf : \mathsf{Intf}$ *and* $\vdash adecs\ \mathsf{ok}$, *then* $adecs \vdash intf : \mathsf{Intf}$.

4. *If* $udecs \vdash sdecs; tdecs \rightsquigarrow intf : \mathsf{Intf}$ *and* $\vdash udecs, sdecs, tdecs\ \mathsf{ok}$, *then* $udecs \vdash decs$ *and* $decs \vdash intf : \mathsf{Intf}$.

5. *If* $decs \vdash impl_0 : intf_0 \preceq intf \rightsquigarrow impl$, *then* $decs \vdash impl_0 : intf_0$, $decs \vdash intf_0 \leq intf : \mathsf{Intf}$, *and* $decs \vdash impl : intf$.

6. *If* $udecs \vdash topdec \rightsquigarrow sbnds : (sdecs; tdecs)$ *and* $\vdash udecs\ \mathsf{ok}$, *then* $udecs \vdash decs$, $decs \vdash sbnds : sdecs$, *and* $\vdash udecs, sdecs, tdecs\ \mathsf{ok}$.

7. *If* $udecs \vdash topspec \rightsquigarrow sdecs; tdecs$ *and* $\vdash udecs\ \mathsf{ok}$, *then* $udecs \vdash decs$, $decs \vdash sdecs\ \mathsf{ok}$, *and* $\vdash udecs, sdecs, tdecs\ \mathsf{ok}$.

8. *If* $udecs \vdash sigbind \rightsquigarrow tdecs$ *and* $\vdash udecs\ \mathsf{ok}$, *then* $\vdash udecs, tdecs\ \mathsf{ok}$.

9. *If* $udecs \vdash sigexp \rightsquigarrow sig : \mathsf{Sig}$ *and* $\vdash udecs\ \mathsf{ok}$, *then* $udecs \vdash decs$ *and* $decs \vdash sig : \mathsf{Sig}$.

10. *If* $udecs \vdash funspec \rightsquigarrow sdecs$ *and* $\vdash udecs\ \mathsf{ok}$, *then* $udecs \vdash decs$ *and* $decs \vdash sdecs\ \mathsf{ok}$.

11. *If* $udecs \vdash_{\mathsf{ctx}} sigid \rightsquigarrow sig : \mathsf{Sig}$ *and* $\vdash udecs\ \mathsf{ok}$, *then* $udecs \vdash decs$ *and* $decs \vdash sig : \mathsf{Sig}$.

12. *If* $adecs \vdash \mathsf{open}\ unitids \rightsquigarrow udecs, \sigma$ *and* $\vdash adecs\ \mathsf{ok}$, *then* $\vdash udecs\ \mathsf{ok}$.

13. *If* $udecs \vdash decs$ *and* $\vdash udecs\ \mathsf{ok}$, *then* $\vdash decs\ \mathsf{ok}$.

Conjecture 22. *The realization of the linker for TS is suitable:*

1. *If* $\vdash assm\ \mathsf{complete}$, *then there exists a program prog such that* $\vdash assm \rightsquigarrow prog$.

2. *If* $\vdash intf\ \mathsf{requires}\ unitid$ *and* $adecs \vdash intf : \mathsf{Intf}$, *then* $unitid \in \mathrm{dom}(adecs)$.

3. *If* $\vdash impl\ \mathsf{requires}\ unitid$ *and* $adecs \vdash impl : intf$, *then* $unitid \in \mathrm{dom}(adecs)$.

4. *If* $\vdash assm \rightsquigarrow prog$ *and* $\vdash assm\ \mathsf{complete}$, *then* $\vdash prog\ \mathsf{ok}$.

5. *If* $\vdash assm \rightsquigarrow bnds : decs$ *and* $\vdash assm\ \mathsf{complete}$, *then* $\vdash bnds : decs$.

F.3 Suitability of the TD Realization

Conjecture 23. *The realization of the IL static semantics for TD is suitable:*

1. *If* $\Gamma \vdash intf : \mathsf{Intf}$, *then* $\vdash \Gamma\ \mathsf{ok}$.

2. *If* $\Gamma \vdash impl : intf$, *then* $\Gamma \vdash intf : \mathsf{Intf}$.

3. *If* $\Gamma \vdash intf \equiv intf' : \mathsf{Intf}$, *then* $\Gamma \vdash intf : \mathsf{Intf}$ *and* $\Gamma \vdash intf' : \mathsf{Intf}$.

4. *If* $\Gamma \vdash intf \leq intf' : \mathsf{Intf}$, *then* $\Gamma \vdash intf : \mathsf{Intf}$ *and* $\Gamma \vdash intf' : \mathsf{Intf}$.

5. *If* $adecs \vdash \Gamma$, *then* $\vdash adecs\ \mathsf{ok}$ *and* $\vdash \Gamma\ \mathsf{ok}$.

6. *If* $\Gamma \vdash B \Rightarrow intf : \mathsf{Intf}$, *then* $\Gamma \vdash intf : \mathsf{Intf}$.

7. *If* $\Gamma \vdash \mathit{intf} \Rightarrow B, L$, *then* $\Gamma \vdash \mathit{intf} : \mathsf{Intf}$, $\vdash B$ ok, $\text{tyvars } B = \emptyset$, $\text{tynames } B \subset T \text{ of } \Gamma$, $(T \text{ of } B) \cap (T \text{ of } \Gamma) = \emptyset$, *and* $\text{tynames } L \subset (T \text{ of } B)$.

8. *If* $\Gamma \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \Rightarrow B$, *then* $\vdash \Gamma$ ok, $\mathit{unitid}_1, \ldots, \mathit{unitid}_n \in \mathrm{dom}(\Gamma)$, $\vdash B$ ok, *and* $\text{tyvars } B = \emptyset$.

Conjecture 24. *The realization of the elaborator for TD is suitable:*

1. $\vdash \mathit{intf}_{\mathit{basis}} : \mathsf{Intf}$.

2. *If* $\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topdec} \rightsquigarrow \mathit{impl} : \mathit{intf}$, *and* $\vdash \mathit{adecs}$ ok, *then* $\mathit{adecs} \vdash \mathit{impl} : \mathit{intf}$.

3. *If* $\mathit{adecs} \vdash \mathsf{open}\ \mathit{unitids}\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf}$ *and* $\vdash \mathit{adecs}$ ok, *then* $\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$.

4. *If* $B \vdash \mathit{topspec} \Rightarrow B'$ *and* $\vdash B$ ok, *then* $\vdash B'$ ok *and* $\text{tyvars } B' = \emptyset$.

5. *If* $B \vdash \mathit{funspec} \Rightarrow F$ *and* $\vdash B$ ok, *then* $\vdash F$ ok.

6. *If* $\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{impl}$, *then* $\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0$, $\Gamma \vdash \mathit{intf}_0 \leq \mathit{intf} : \mathsf{Intf}$, *and* $\Gamma \vdash \mathit{impl} : \mathit{intf}$.

Conjecture 25. *The realization of the linker for TD is suitable:*

1. *If* $\vdash \mathit{assm}$ $\mathsf{complete}$, *then there exists a program prog such that* $\vdash \mathit{assm} \rightsquigarrow \mathit{prog}$.

2. *If* $\vdash \mathit{intf}$ $\mathsf{requires}$ unitid *and* $\mathit{adecs} \vdash \mathit{intf} : \mathsf{Intf}$, *then* $\mathit{unitid} \in \mathrm{dom}(\mathit{adecs})$.

3. *If* $\vdash \mathit{impl}$ $\mathsf{requires}$ unitid *and* $\mathit{adecs} \vdash \mathit{impl} : \mathit{intf}$, *then* $\mathit{unitid} \in \mathrm{dom}(\mathit{adecs})$.

4. *If* $\vdash \mathit{assm} \rightsquigarrow \mathit{prog}$ *and* $\vdash \mathit{assm}$ $\mathsf{complete}$, *then* $\vdash \mathit{prog}$ ok.