A Separate Compilation Extension to Standard ML (Revised and Expanded)

David Swasey
Carnegie Mellon University

Tom Murphy VII
Carnegie Mellon University

Karl Crary
Carnegie Mellon University

Robert Harper
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci

Published In
Proceedings of the 2006 Workshop on ML, ML '06., 32-42.
Abstract

We present an extension to Standard ML, called SMLSC, to support separate compilation. The system gives meaning to individual program fragments, called units. Units may depend on one another in a way specified by the programmer. A dependency may be mediated by an interface (the type of a unit); if so, the units can be compiled separately. Otherwise, they must be compiled in sequence. We also propose a methodology for programming in SMLSC that reflects code development practice and avoids syntactic repetition of interfaces. The language is given a formal semantics, and we argue that this semantics is implementable in a variety of compilers.
Keywords: Standard ML, separate compilation, incremental compilation, types
Introduction

We propose an extension to Standard ML called SMLSC. SMLSC supports separate compilation in the sense that it gives a static semantics to individual program fragments, which we call units. A unit may depend on other units, and can be type-checked independently of those units by specifying what it expects of them. These expectations are given in the form of interfaces for those other units. When unit $A$ is checked against another unit $B$ via a mediating interface, we need not have access to $B$ at all. Therefore we say that $A$ is separately compiled (SC) against $B$.

It is also useful to allow unit $A$ to depend on another unit $B$ without specifying an interface for $B$. In this case, the only way to derive the context necessary to check $A$ is to first check $B$ and read off its actual interface. In this scenario we say that $A$ is incrementally compiled (IC) against $B$.

Units may be compiled and then linked together to satisfy dependencies. The compiled form of a unit or set of linked units is called a linkset. A linkset may be further linked with other linksets. If a linkset has no remaining dependencies, then it can be transformed into an executable program.

The goal of this work is to consolidate and synthesize previous work on compilation management for ML into a formally defined extension to the Standard ML language. The extension itself is syntactically and conceptually very simple. A unit is a series of Standard ML top-level declarations, given a name. To the current top-level declarations such as `structure` and `functor` we add an `import` declaration that is to units what the `open` declaration is to structures. An `import` declaration may optionally specify an interface for the unit, in which case we are able to compile separately against that dependency; with no interface we must compile incrementally. Compatibility with existing compilers, including whole-program compilers, is assured by making no commitment to the precise meaning of “compile” and “link”—a compiler is free to limit compilation to elaboration and type checking, and to perform code generation as part of linking.

Sections 1 and 2 summarize our main design principles, and provide an overview of the system. In Section 2 we give a small example of its use. (We give a larger example in Appendix G.) The semantics, formulated in the framework of the Harper-Stone semantics of ML [12, 13], is given in Section 3. We give an alternative semantics in the framework of The Definition of Standard ML [16] in Section 4. Some implementation issues are discussed in Section 5. We conclude with a discussion of related work in Section 7.

1 Design Principles

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 Standard ML 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 declarations, including signature and functor declarations. However, since Standard ML lacks syntactically expressible principle signatures, some units cannot be separately compiled from one another. We therefore support incremental, as well as separate, compilation for any unit. This means that the interface of a unit can either be inferred from its source (incremental compilation) or explicitly specified (separate compilation) at the programmer’s discretion.

---

1 Consequently, units cannot be identified with Standard ML structures.
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 additional machinery.

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 Standard ML. This ensures that compilers should not be disturbed by the proposed extension beyond what is required to implement the extension itself.

Explicit dependencies. The dependencies among units are 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. There are in general many ways to order effects consistently with syntactic 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.

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.

Separation of units from modules. The separate compilation system is designed as a proper extension to Standard ML so as to ensure backward compatibility of source code. It is tempting to identify compilation units with modules, but to do so would require that functors, signatures, and fixity declarations be permitted as components of modules. Permitting such an extension is not entirely straightforward; for example, permitting signature declarations in modules and their types can lead to undecidability of type checking [10].

2 Overview

Units and Interfaces

The SMLSC extension is organized around the concept of a unit. A unit consists of top-level declarations, which include declarations of signatures, structures, and functors. Each unit is given a name by which the unit is known throughout the program. One unit may refer to the components of another using an import declaration, which records the dependency of the importing unit on the imported unit, and opens it for use within the importing unit. This is the only means by which one unit may refer to another; we do not support “dot notation” for accessing the components of a unit. An import declaration is a new form of top-level declaration. (This is the only modification that we make to an existing syntactic category of Standard ML.)

The compilation context for a unit is entirely determined by its imports. That is, all dependencies of a unit on another unit must be explicitly indicated using import declarations. The dependency of one unit on another is mediated by an interface, the type of a unit. The interface of an imported unit can be specified in one of two ways, either implicitly or explicitly, corresponding to incremental or separate compilation.

An import declaration of the form import unitid : intexp specifies an explicit interface for the imported unit. This permits the importing unit to be compiled independently of the implementation of unitid, relying only on the specified interface. This is called separate compilation, or SC for
short. An import declaration of the form `import unitid` specifies that the interface for `unitid` is to be inferred from its source code. This is called incremental compilation, or IC for short.

The concrete syntax for units and interfaces in SMLSC is given in Figure 1. We extend `topdec`s to add `import` and `local`. The `import` declaration (like `open`) allows multiple units to be simultaneously imported. Interfaces are `topspec`s; this is the syntactic class `spec` of Standard ML, with the addition of a specification form for functors. The `local` declaration limits the scope of imports, just as the structure declaration of the same name.

Projects and Linksets

A `linkset` consists of several compiled units, called its exports, together with the names and interfaces of its imports, the units on which it depends (following Cardelli [5]). A `project` consists of a linearly ordered sequence of source units and linksets. The ordering of the components in a project is significant, both because it specifies the order of identifier resolution, and because it specifies the order of computational effects when executed. Compilation of a project consists of processing the source units in the specified order to obtain linksets, and then knitting them together to resolve dependencies.

Linking consists of resolving inter-unit dependencies by binding exports to imports among linksets. When all references have been resolved, the resulting linkset can be completed to form an executable.

We do not give a concrete syntax for linksets, as we do not intend for programmers to write them, nor do we expect compatibility of linksets across implementations. Rather, they are left as implementation-specific concepts (such as object files), which are modeled here by the abstract semantic objects described in Sections 3 and 4.

Examples

We begin with a few simple examples to illustrate the features of the system.

Suppose that we have a library of data structures whose name is Collections. It is natural to place this library in a unit. Let’s assume it contains only the queue data structure:
unit Collections =
unit
signature QUEUE =
sig
  type 'a queue
  val empty : 'a queue
  val push : 'a * 'a queue -> 'a
  val pop : 'a queue -> 'a * 'a queue
end

structure Queue :> QUEUE =
  struct (* · · · *) end
end

A client of the Collections library can import it using IC easily:

unit Scheduler =
unit
import Collections

structure Sched =
  struct
    type job = (* · · · *)
    val readyqueue =
      ref Queue.empty : job Queue.queue ref
      (* · · · *)
  end
end

In these examples we use link to stand for the semantic operation of compiling and linking a list of source units and linksets. We can compile and link this program as

\[ L = link(Collections, Scheduler) \]

or we can compile the library and then the client

\[ L_0 = link(Collections) \]
\[ L_1 = link(L_0, Scheduler) \]

but the Scheduler unit may not be compiled on its own.

Incremental compilation is convenient when we have source or a compiled linkset for the Collections unit. We may prefer to use separate compilation, or may be forced to because the implementation for Collections is not available. A client with an SC import looks like this:
unit Scheduler2 =
unit
import Collections :
intf
structure Queue :
sig
type 'a queue
val empty : 'a queue
val push : 'a * 'a queue -> 'a
val pop : 'a queue -> 'a * 'a queue
end
end
structure Sched =
struct
type job = (* ... *)
val readyqueue =
  ref Queue.empty : job Queue.queue ref
(* ... *)
end
end

This allows Scheduler2 and Collections to be compiled separately:

\[ L_0 = \text{link}(Scheduler2) \]
\[ L_1 = \text{link}(Collections) \]
\[ L_2 = \text{link}(L_1, L_0) \]

However, writing the SC import this way forces an undesirable repetition of code. If more than one client uses Collections—which we would expect—each client repeats the interface for its import of the unit. A further problem is that this style asks the client to supply the interface of the library, but the interface of a library is usually provided by the library author, not the client. Fortunately, a combination of SC and IC allows us to use the system in a much cleaner way.

Handoff Units

A programmer who wishes his code to be available for separate compilation can provide a handoff unit which supplies the interface. Starting from scratch, the handoff unit contains an SC import:

unit Collections =
unit
signature QUEUE =
sig
type 'a queue
val empty : 'a queue
val push : 'a * 'a queue -> 'a
val pop : 'a queue -> 'a * 'a queue
end
The implementation of the collections library is imported from the unit CollectionsImpl. Because the import declaration opens the imported unit, all of the contents of CollectionsImpl are available in the Collections unit. Clients wishing to make use of the library simply import the handoff unit using IC, avoiding the need to specify an interface, but instead sharing the common interface provided by the handoff unit.

```
unit Scheduler3 =
unit
  import Collections

  structure Sched = (* · · · *)
end
```

This additionally has the benefit that the clients only need to know the name of the handoff unit, not the implementation unit. A few such clients can be linked with the handoff unit:

\[ L_0 = \text{link}(\text{Collections}, \text{Scheduler3}, \text{OtherClient}) \]

The result can later be linked with the implementation of the Collections library:

\[ L_1 = \text{link}(\text{CollectionsImpl}, L_0) \]

**Definite References**

In the terminology of Harper and Pierce [11] an import of one unit in another is interpreted as a definite reference—that is, as a free variable that refers to a single, specific unit through an interface for it, either inferred or specified. This ensures that if two separate units import a common unit, such as a well-known library, these units share a common understanding of the abstract types exported by that unit.

In contrast, the fully functorized style is to \( \lambda \)-abstract a module over all of the modules on which it depends. Because functors may, in principle, be applied to many different arguments, its parameters are indefinite references. As such sharing relationships among components are lost, because they need not be true in every instance. To avoid this, one must explicitly specify the intended sharing constraints; this can be quite burdensome.

The handoff methodology in SMLSC facilitates programming with definite references. Two pieces of code that import the same unit using separate compilation may only be linked if they import that unit at equivalent interfaces. The imports are then consolidated into a single import, ensuring that type equations hold. When the same handoff unit is used to create the two imports, these interfaces will always be equivalent. In corner cases such as skew between versions of a library’s handoff unit, the programmer may manually consolidate two imports. We discuss this further in Section 6.

---

2 If unit CollectionsImpl also needs signature QUEUE, it can be placed in its own unit QUEUESIG and both CollectionsImpl and Collections can incrementally compile against QUEUESIG. Appendix G exemplifies this approach.
3 Semantics extending the Harper-Stone semantics of ML

In this section, we give a semantics to SMLSC by extending Harper and Stone’s Typed Semantics (TS) for Standard ML [13]. At a high level the typed semantics consists of an elaboration relation from an external language, called TSEL, into an internal language, called TSIL. The external language is a slight extension of the abstract syntax of Standard ML. The internal language is a typed λ-calculus based on the Harper-Lillibridge type theory for modules [10]. Elaboration comprises type inference, pattern compilation, equality compilation, identifier resolution, and insertions of coercions for signature matching. The result of elaboration is a well-formed program in the TSIL, to which a dynamic semantics is given to provide an execution model. The semantics of SMLSC is an extension of the Harper-Stone semantics that elaborates units into linksets that can be completed for execution.

The TSIL. We begin with a brief review of the structure of the TSIL. The TSIL consists of a core level and a module level. The core level includes expressions exp, constructors con, and kinds kend. Kinds classify constructors. Constructors of kind Ω are types; they classify expressions. The module level includes modules mod and signatures sig, which classify modules. We write {} to denote the empty record, and mod.lab to denote the projection of a component named lab from the structure mod. The semantics works mainly with modules, ultimately elaborating units to TSIL structures.

Declaration lists serve as contexts in the TSIL static semantics. A declaration list dece = dec1, . . . , decn declares expression (var:con), constructor (var:knd(=con)), and module (var:sig) variables. A structure declaration list sdecs has the form

lab1▷dec1, . . . , labn▷decn

associating a label with each declaration. The structure declaration list lab▷dec, sdecs binds the variable declared by dec with scope sdecs. We write [sdecs] to denote the signature of a structure containing fields described by sdecs. Variables express dependencies between components in a structure signature and may be freely alpha-varied. Labels name components for external reference and may not be renamed without changing the meaning of the signature. Consider the declaration of a structure m containing an opaque type component T and value component X of that type:

m : [T▷t:Ω, X▷x:t].

We can systematically rename the bound variables t and x. A path is a module variable followed by a list of labels, serving a role similar to SML long identifiers. The paths m.T (a constructor) and m.X (an expression) refer to m’s components.

A bnd binds a variable to an expression (var=exp), constructor (var=con), or module (var=mod). A structure binding list sbnds has the form

lab1▷bnd1, . . . , labn▷bndn.

A structure is written [sbnds]. The module syntax is closed under the formation of functors: dependently typed functions from modules to modules.

We shall use the TSIL judgements given in Figure 2. These judgements have the following meaning.

- dece ⊢ sdecs ok. No label is used twice and every declaration is well-formed. For example,

  ⊢ T▷t:Ω, X▷x:t ok.
<table>
<thead>
<tr>
<th>Judgement...</th>
<th>Meaning...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash \text{decs ok} )</td>
<td>decs is well-formed</td>
</tr>
<tr>
<td>decs ( \vdash \text{sdecs ok} )</td>
<td>sdecs is well-formed</td>
</tr>
<tr>
<td>decs ( \vdash \text{sig : Sig} )</td>
<td>sig is well-formed</td>
</tr>
<tr>
<td>decs ( \vdash \text{sig \equiv sig' : Sig} )</td>
<td>signature equivalence</td>
</tr>
<tr>
<td>decs ( \vdash \text{sbnbs : sdecs} )</td>
<td>sbnbs has declaration list sdecs</td>
</tr>
<tr>
<td>decs ( \vdash \text{mod : sig} )</td>
<td>mod has signature sig</td>
</tr>
</tbody>
</table>

Figure 2: TSIL judgements (summary)

<table>
<thead>
<tr>
<th>Judgement...</th>
<th>Meaning...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash \text{strdec \Rightarrow sbnds : sdecs} )</td>
<td>structure declaration elaboration</td>
</tr>
<tr>
<td>( \Gamma \vdash \text{sigexp \Rightarrow sig : Sig} )</td>
<td>signature elaboration</td>
</tr>
<tr>
<td>( \Gamma \vdash \text{spec \Rightarrow sdecs} )</td>
<td>specification elaboration</td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ctx, labs \Rightarrow path : class} )</td>
<td>context lookup</td>
</tr>
<tr>
<td>decs ( \vdash \text{sub : sig \leq sig \Rightarrow mod : sig'} )</td>
<td>coercion compilation</td>
</tr>
</tbody>
</table>

Figure 3: TS elaboration judgements (summary)

- decs \( \vdash \text{sig : Sig} \). The signature sig is well-formed.
- decs \( \vdash \text{sig \equiv sig' : Sig} \). The signatures sig and sig' declare the same components, in the same order, with the same labels, and corresponding type components are equivalent.
- decs \( \vdash \text{sbnbs : sdecs} \). The structure binding list sbnbs matches the structure declaration list sdecs. Corresponding labels must agree and each bound expression, constructor, or module in sbnbs must match its declaration in sdecs. For example, the judgement

\[
\text{decs} \vdash (\text{lab>var=} \text{mod}, \text{sbnbs}) : (\text{lab>var: sig, sdecs})
\]

holds if decs \( \vdash \text{mod : sig} \) and decs, var: sig \( \vdash \text{sbnbs : sdecs} \).
- decs \( \vdash \text{mod : sig} \). The module mod has signature sig. The signature sig may or may not be fully transparent. For example, we may derive both

\[
m : [T \vdash t : \Omega, X \vdash x : t] \vdash m : [T \vdash t : \Omega \Rightarrow m.T, X \vdash x : t]
\]

and

\[
m : [T \vdash t : \Omega, X \vdash x : t] \vdash m : [T \vdash t : \Omega, X \vdash x : t].
\]

The former signature is said to be selfified with respect to the variable m.

**TS elaboration.** Harper and Stone give a semantics to Standard ML by elaboration of TSEL into TSIL. Elaboration is performed in a context \( \Gamma \) consisting of a structure declaration list (sdecs) that, due to shadowing, may have duplicate labels. We shall use the TS elaboration judgements given in Figure 3. These judgements have the following meaning.
• $\Gamma \vdash \text{strdec} \rightsquigarrow \text{sbnds} : \text{sdecs}$. Elaborate the TSEL structure declaration \text{strdec} to the structure binding list \text{sbnds} : \text{sdecs}. Since the TSEL permits functors within structures, this includes elaboration of functor declarations.

• $\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}$. Elaborate the TSEL signature expression \text{sigexp} to the signature \text{sig}. The TSEL does not include signature declarations; we treat them as abbreviations for TSIL signatures, recording them in linksets and expanding them during elaboration.

• $\Gamma \vdash \text{spec} \rightsquigarrow \text{sdecs}$. Elaborate the TSEL specification \text{spec} to the structure declaration list \text{sdecs}. This includes elaboration of functor specifications.

• $\Gamma \vdash_{\text{ctx}} \text{labs} \rightsquigarrow \text{path} : \text{class}$. Perform identifier resolution in the context $\Gamma$. The input is a list of labels, which is derived from an SML long identifier; the output is a path classified by the type, kind, or signature \text{class}. Some labels in the context are annotated with a star, indicating that they are “open” (in the sense of the SML \text{open} declaration). Identifier resolution searches $\Gamma$ from right to left, descending into structures with starred labels. For example, we may derive

$$T \triangleright t_1 : \Omega, T \triangleright t_2 : \Omega = \{ \} \vdash \text{spec} \rightsquigarrow t_2 : \Omega = \{ \}$$

and

$$X \triangleright x_1 : \{ \}, 1^* \triangleright m : [T \triangleright t : \Omega, X \triangleright x_2 : t] \vdash X \rightsquigarrow m.X : m.T.$$

• $\text{decs} \vdash_{\text{sub}} \text{path} : \text{sig}_0 \preceq \text{sig} \rightsquigarrow \text{mod} : \text{sig}'$. Perform transparent signature ascription. The inputs are a signature \text{sig}_0, a \text{path} having that signature, and a target signature \text{sig}. The output is a module \text{mod} : \text{sig}', where \text{sig}' has the same shape as \text{sig} but is fully transparent relative to \text{path}.

Elaboration maps TSEL identifiers to TSIL labels using a function $\tau$. To implement identifier “shadowing,” elaboration employs a function $\text{sbnds} \leftrightarrow \text{sbnds}' : \text{sdecs} \leftrightarrow \text{sdecs}'$ that concatenates \text{sbnds} : \text{sdecs} and \text{sbnds}' : \text{sdecs}', renaming labels in the left hand sides that appear in the right hand sides. The function chooses fresh labels that do not correspond to TSEL identifiers. For example, if

$$\text{sbnds} : \text{sdecs} = T \triangleright t_1 = \{ \}, T \triangleright t_1 : \Omega = \{ \}$$
$$\text{sbnds}' : \text{sdecs}' = T \triangleright t_2 = \text{Int} : T \triangleright t_2 : \Omega = \text{Int},$$

then $\text{sbnds} \leftrightarrow \text{sbnds}' : \text{sdecs} \leftrightarrow \text{sdecs}'$ might be

$$(\text{lab} \triangleright t_1 = \{ \}, T \triangleright t_2 = \text{Int}) : (\text{lab} \triangleright t_1 : \Omega = \{ \}, T \triangleright t_2 : \Omega = \text{Int})$$

where \text{lab} is not in the range of the $\tau$ function.

3.1 Linking

We define linking for the TSIL by giving rules for deriving the linking judgements in Figure 4. A linkset

$$\text{sdecs}_0 \rightarrow \text{sbnds} : \text{sdecs}; S$$

comprises imports $\text{sdecs}_0$, exports $\text{sbnds} : \text{sdecs}$, and signature abbreviations $S$. 
<table>
<thead>
<tr>
<th>Judgement...</th>
<th>Meaning...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decs ⊢ L ok</code></td>
<td><code>L</code> is well-formed</td>
</tr>
<tr>
<td><code>decs ⊢ S ok</code></td>
<td><code>S</code> is well-formed</td>
</tr>
<tr>
<td><code>L ⇝ exp : {}</code></td>
<td><code>L</code> completes to <code>exp</code></td>
</tr>
<tr>
<td><code>decs ⊢ L ++ L' ⇝ L''</code></td>
<td><code>L</code> and <code>L'</code> merge to <code>L''</code></td>
</tr>
</tbody>
</table>

Figure 4: Linking judgements

\[
L ::= sdecs_0 \rightarrow sbnds : sdecs; S \quad \text{linkset}
\]
\[
S ::= .
\]
\[
S, \text{sigid} = \text{sig} \quad \text{top-level}
\]
\[
S, \text{unitid} = S' \quad \text{declared by unitid}
\]

Figure 5: Linkset syntax

- The imports `sdecs_0` describe the TSIL structures on which the linkset depends; they must be well-formed in the ambient context. For example, the imports

\[
sdecs_{AB} = A \triangleright a : [T \triangleright t : \Omega, X \triangleright x : t],\]
\[
B \triangleright b : [Y \triangleright y : a.T]
\]

express dependency on structures labelled `A` and `B`.

Imports specify assumptions to be satisfied by linking. A linkset with imports `sdecs_{AB}` assumes structure `B` binds (at least) a value `Y : a.T` but can be linked with (a linkset exporting) a structure `B` providing more components.

- The exports `sbnds : sdecs` are the TSIL code associated with the linkset. They may make reference to the linkset’s imports. Continuing our example, the exports

\[
sbnds_{ZR} : sdecs_{ZR} = Z \triangleright z = b.Y, R \triangleright r = a.T : Z \triangleright z : a.T, R \triangleright r : \Omega = a.T
\]

reference the imports `sdecs_{AB}` to bind an expression `Z` of the imported type and an equivalent type `R`.

- The signature abbreviations `S` are used during elaboration. They may make reference to the linkset’s imports and exports. Continuing our example, the signature abbreviations

\[
S_{\text{SIG}} = \text{SIG} = [M \triangleright m : \Omega = r]
\]

specify that elaboration should treat the signature identifier `SIG` as an abbreviation for a TSIL signature referencing the exported type `R`.

The dynamic semantics for SMLSC is very simple. The completion judgment `L ⇝ exp : {}` translates a linkset

\[
\cdot \rightarrow sbnds : sdecs; S
\]

with no imports to a TSIL expression

\[
[sbnds, \text{lab} \triangleright \text{var} = \{\}].\text{lab} : \{}
\]
where \textit{lab} and \textit{var} are fresh. Under the TSIL dynamic semantics, the resulting expression evaluates the linkset’s exports from left to right for their side-effects. Evaluation terminates when an uncaught exception is raised or when every export has been evaluated.

We give the full syntax for linksets in Figure 5 and the rules in Appendix A. The remainder of this section explains the rules for linkset merge.

\textbf{Notation.} We write \textit{decs}, \textit{sdecs} to extend a context \textit{decs}, implicitly dropping the labels in \textit{sdecs}. We define the domain of a structure declaration list, \textit{dom(sdecs)}, by

\[
\text{dom}(\text{lab}_1 \triangleright \text{dec}_1, \ldots, \text{lab}_n \triangleright \text{dec}_n) = \{\text{lab}_1, \ldots, \text{lab}_n\}.
\]

We write \{var/var’\}\textit{L} for the capture-free substitution of var for free occurrences of var’ in \textit{L}.\(^3\)

\textbf{Linkset merge.} The rules for linkset merge \textit{decs} \vdash L_1 ++ L_2 \leadsto L_3 combine \textit{L}_1 and \textit{L}_2 to produce \textit{L}_3. The rules presuppose that \textit{L}_1 is well-formed with respect to \textit{decs} but permit \textit{L}_2 to make reference (via free TSIL variables) not only to \textit{decs} but to the imports and exports of \textit{L}_1. Formally, the rules satisfy the following property.\(^4\)

\[
\text{If } L_1 = \text{sdecs}_1 \rightarrow \text{sbnds} : \text{sdecs} \quad \text{and } \text{decs} \vdash L_1 \text{ ok} \\
\text{and } \text{decs, sdecs}_1, \text{sdecs} \vdash L_2 \text{ ok}, \\
\text{and } \text{decs} \vdash L_1 ++ L_2 \leadsto L_3, \\
\text{then } \text{decs} \vdash L_3 \text{ ok}.
\]

If a linkset is well-formed, then it neither imports nor exports the same label twice (although it may both import and export a particular label).

The rules process the imports in \textit{L}_2 from left to right. If \textit{L}_2 has no imports, then the following rule applies.

\[
\begin{align*}
L = \text{sdecs}_0 \rightarrow \text{sbnds} : \text{sdecs} \\
\text{decs} \vdash L'++(\cdot \rightarrow \text{sbnds}' : \text{sdecs}') \leadsto \\
\text{sdecs}_0 \rightarrow \text{sbnds}++\text{sbnds}' : \text{sdecs}++\text{sdecs}'
\end{align*}
\]

\textit{L}_3 imports what \textit{L}_1 does and exports what \textit{L}_1 and \textit{L}_2 do. To ensure that \textit{L}_3 is well-formed, the rule uses the TS function ++ to concatenate the exports in \textit{L}_1 with the exports in \textit{L}_2, renaming labels exported by \textit{L}_1 that are also exported by \textit{L}_2.

Otherwise, the rules examine the first import \textit{lab>var:sig} in \textit{L}_2 and distinguish three mutually exclusive cases:

\begin{itemize}
\item \textit{L}_1 exports \textit{lab}.
\end{itemize}

\[
\begin{align*}
\text{sdecs} = \text{sdecs}''', \text{lab>var'}:\text{sig'}, \text{sdecs}'' & \\
\text{decs, sdecs}_0, \text{sdecs} \vdash_{\text{sub}} \text{var'}:\text{sig'} \preceq \text{sig} \leadsto \text{mod} : \text{sig}'' & \\
\text{sbnd} := 1\triangleright \text{var} = \text{mod} & \\
\text{sdec} := 1\triangleright \text{var} : \text{sig}'' & \\
L := \text{sdecs}_0 \rightarrow \text{sbnds}++\text{sbnd} : \text{sdecs}++\text{sdec} & \\
\text{decs} \vdash L'++(\text{sdecs}_1 \rightarrow \text{sbnds}' : \text{sdecs}') \leadsto L'' & \\
\text{decs} \vdash (\text{sdecs}_0 \rightarrow \text{sbnds} : \text{sdecs})++ \\
(\text{lab>var} : \text{sig}, \text{sdecs}_1 \rightarrow \text{sbnds}' : \text{sdecs}') \leadsto L''
\end{align*}
\]

\(^3\)Linkset bound variables and scopes are discussed in Appendix A.

\(^4\)In this description of linkset merge, we suppress all details related to signature abbreviations.
The first premise picks out the $L_1$ export $lab \triangleright var' : sig'$ for $lab$; there can be at most one since $L_1$ is well-formed. The second premise calls the TS coercion compiler to match the export $var' : sig'$ to the import signature $sig$. Linking fails if no match is possible; otherwise, $sig''$ has the same “shape” as $sig$, but is fully transparent relative to the variable $var'$. The structure binding $sbnd : sdec$ is constructed using the coercion module $mod$ at the signature $sig''$, maximizing type sharing. The linkset $L$ has the same imports as $L_1$, and exports those of $L_1$ plus the result of the preceding coercion. To ensure that $L$ is well-formed—in particular, that it exports nothing more than once—the rule uses $++$ to construct its exports.

- $L_1$ imports $lab$ but does not export it.

\[
lab \notin \text{dom}(sdecs)
\]
\[
sdecs_0 = sdecs'', lab \triangleright var' : sig', sdecs''
\]
\[
decs, sdecs_0, sdecs \vdash sig \equiv sig' : \text{Sig}
\]
\[
L' := \{var'/var\}(sdecs_1 \rightarrow sbnds' : sdecs')
\]
\[
decs \vdash (sdecs_0 \rightarrow sbnds : sdecs)++L' \rightsquigarrow L''
\]
\[
decs \vdash (sdecs_0 \rightarrow sbnds : sdecs)++(lab \triangleright var : sig, sdecs_1 \rightarrow sbnds' : sdecs') \rightsquigarrow L''
\]

The first premise ensures $L_1$ does not export $lab$. The second premise picks out the $L_1$ import $lab \triangleright var' : sig'$. Linking fails if $sig$ and $sig'$ are not equivalent; otherwise, $L'$ is constructed by changing references in the remainder of $L_2$ to use the import in $L_1$.

- $L_1$ neither imports nor exports $lab$.

\[
lab \notin \text{dom}(sdecs) \cup \text{dom}(sdecs_0)
\]
\[
decs, sdecs_0, sdecs \vdash sig \equiv sig' : \text{Sig}
\]
\[
L := sdecs_0, lab \triangleright var : sig' \rightarrow sbnds : sdecs
\]
\[
decs \vdash L++(sdecs_1 \rightarrow sbnds' : sdecs') \rightsquigarrow L''
\]
\[
decs \vdash (sdecs_0 \rightarrow sbnds : sdecs)++(lab \triangleright var : sig, sdecs_1 \rightarrow sbnds' : sdecs') \rightsquigarrow L''
\]

The first premise ensures that $L_1$ neither imports nor exports $lab$. The next two premises choose a signature $sig'$ equivalent to $sig$ but well-formed without reference to the exports of $L_1$. Linking fails if no such signature exists—when opaque types exported by $L_1$ occur in $sig$. Otherwise, $L$ is constructed by adding a new import to the imports in $L_1$.

### 3.2 Elaboration

We define a semantics for SMLSC by giving rules for the elaboration judgements in Figure 6. We give the abstract syntax for SMLSC in Figure 7. The elaboration rules appear in Appendix B. These judgements have the following meaning.

- **project $\rightsquigarrow L$**. Elaborate $project$, using linkset merge to accumulate a resulting linkset $L$. A source unit is elaborated in a context $\Gamma$ that declares the imports and exports in $L$.

- **$\Gamma \vdash srcunit \rightsquigarrow L$**. Elaborate the $\text{topdec}$ in $srcunit$ to the linkset

\[
sdecs_0 \rightarrow sbnds : sdecs; S.
\]
### Elaboration Judgements

<table>
<thead>
<tr>
<th>Judgement...</th>
<th>Meaning...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>project ⇝ L</code></td>
<td>project elaboration</td>
</tr>
<tr>
<td><code>Γ ⊢ srcunit ⇝ L</code></td>
<td>unit elaboration</td>
</tr>
<tr>
<td><code>Γ ⊢ topdec ⇝ L</code></td>
<td>top-level declaration elaboration</td>
</tr>
<tr>
<td><code>Γ ⊢ impexp ⇝ L</code></td>
<td>import expression elaboration</td>
</tr>
<tr>
<td><code>Γ ⊢ sigbind ⇝ S</code></td>
<td>signature binding elaboration</td>
</tr>
<tr>
<td><code>Γ ⊢\text{ctx} sigid ⇝ sig : \text{Sig}</code></td>
<td>signature lookup</td>
</tr>
<tr>
<td><code>Γ ⊢\text{ctx} unitid ⇝ S</code></td>
<td></td>
</tr>
<tr>
<td><code>Γ \text{ok}</code></td>
<td>Γ is well-formed</td>
</tr>
</tbody>
</table>

Figure 6: Elaboration judgements

### SMLSC Abstract Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>project ::= .</code></td>
<td>empty</td>
</tr>
<tr>
<td><code>project, srcunit</code></td>
<td>source unit</td>
</tr>
<tr>
<td><code>project, L</code></td>
<td>compiled unit(s)</td>
</tr>
<tr>
<td><code>srcunit ::= unit unitid = topdec</code></td>
<td>unit declaration</td>
</tr>
<tr>
<td><code>topdec ::= import impexp</code></td>
<td>open units</td>
</tr>
<tr>
<td><code>strdec</code></td>
<td></td>
</tr>
<tr>
<td><code>signature sigbind</code></td>
<td></td>
</tr>
<tr>
<td><code>local topdec_1 in topdec_2 end</code></td>
<td></td>
</tr>
<tr>
<td><code>topdec_1, topdec_2</code></td>
<td></td>
</tr>
<tr>
<td><code>impexp ::= unitid \langle : \text{intf spec end} \rangle</code></td>
<td>open <code>unitid</code></td>
</tr>
<tr>
<td><code>impexp_1, impexp_2</code></td>
<td></td>
</tr>
<tr>
<td><code>sigbind ::= sigid = sigexp (\text{and sigbind})</code></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: SMLSC abstract syntax
The imports $sdecs_0$ arise from the import declarations in $topdec$. The exports $sbn$ $: sdecs$ arise from the structure declarations in $topdec$. The signature abbreviations $S$ arise from the signature declarations in $topdec$. The result, $L$, exports a single module

\[
\text{unitid} \triangleright \text{var} = [sbn] : \text{unitid} \triangleright \text{var} : [sdecs].
\]

- $\Gamma \vdash \text{topdec} \rightsquigarrow L$. Elaborate $\text{topdec}$ using linkset merge and identifier resolution.
- $\Gamma \vdash \text{impexp} \rightsquigarrow L$. Elaborate $\text{impexp}$ using identifier resolution and $\text{spec}$ elaboration.
- $\Gamma \vdash \text{sigbind} \rightsquigarrow S$. Elaborate $\text{sigbind}$ using signature elaboration.

## 4 Semantics extending The Definition of Standard ML

In this section, we give a semantics to SMLSC by extending The Definition of Standard ML (TD) [16]. TD gives a semantics to SML by relating it to semantic objects—mathematical sets, functions, and so on. We refer to these semantic objects collectively as the internal language (TDIL) and to SML as the external language (TDEL). The TDIL is partitioned into static and dynamic semantic objects. TD’s static semantics specifies type checking and type inference using the static TDIL. TD’s dynamic semantics gives the TDEL a big-step, call-by-value operational semantics using the dynamic TDIL.

A unit declaration list $udecs$ is a list of source units $srcunit_1, \ldots, srcunit_n$ (see Figure 8). In Section 4.1 we review TD’s static semantics and extend it to unit declaration lists. In Section 4.2 we do the same for TD’s dynamic semantics. We define linksets and linking in Section 4.3. A linkset contains source code—a unit declaration list—and static TDIL: Separate compilation corresponds to separate type checking. In Section 4.4 we give a semantics to SMLSC through an elaboration into linksets.

### 4.1 Static Semantics

We shall use the TD static semantic judgements given in Figure 9. These judgements have the following meaning.

- $B \vdash \text{strdec} \Rightarrow E$. The structure declaration $\text{strdec}$ is well-typed and declares the structure, type, and value identifiers in environment $E$.
- $B \vdash \text{sigdec} \Rightarrow G$. The signature declaration $\text{sigdec}$ is well-formed and declares the signature identifiers in signature environment $G$.
- $B \vdash \text{fundec} \Rightarrow F$. The functor declaration $\text{fundec}$ is well-typed and declares the functor identifiers in functor environment $F$.
- $B \vdash \text{sigexp} \Rightarrow \Sigma$. The signature expression $\text{sigexp}$ is well-formed and specifies the components in signature $\Sigma$.

\[
\text{udecs} ::= \cdot \quad \text{empty} \\
\text{udecs}, \text{srcunit} \quad \text{unit declaration}
\]

Figure 8: Unit syntax
<table>
<thead>
<tr>
<th>Judgement...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B \vdash \text{strdec} \Rightarrow E$</td>
</tr>
<tr>
<td>$B \vdash \text{sigdec} \Rightarrow G$</td>
</tr>
<tr>
<td>$B \vdash \text{fundecl} \Rightarrow F$</td>
</tr>
<tr>
<td>$B \vdash \text{sigexp} \Rightarrow \Sigma$</td>
</tr>
<tr>
<td>$B \vdash \text{spec} \Rightarrow E$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Meaning...</th>
</tr>
</thead>
<tbody>
<tr>
<td>structure declaration elaboration</td>
</tr>
<tr>
<td>signature declaration elaboration</td>
</tr>
<tr>
<td>functor declaration elaboration</td>
</tr>
<tr>
<td>signature expression elaboration</td>
</tr>
<tr>
<td>specification elaboration</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\Sigma \geq E$ using $\varphi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>signature instantiation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$E_1 \succ E_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>enrichment</td>
</tr>
</tbody>
</table>

Figure 9: TD’s static semantic judgements (summary)

- $B \vdash \text{spec} \Rightarrow E$. The specification $\text{spec}$ is well-formed and specifies the components in $E$.
- $\Sigma \geq E$ using $\varphi$. The environment $E$ is an instance of the signature $\Sigma$ using the realization $\varphi$.
- $E_1 \succ E_2$. The environment $E_1$ may have more components than $E_2$, it may be less polymorphic, and it may change the status of constructors/exceptions to values.

One subtlety in these judgements pervades our semantics: They account for TDEL type sharing by stamping TDIL types with *type names* and TDEL type generativity by using state-passing to track the set of type names that “have been generated”. Two TDIL types share if they are stamped with the same type name. To see how state-passing works, consider the judgement

$$B \vdash \text{strdec} \Rightarrow E.$$  

The basis $B = T, F, G, E'$ comprises a context and a state. The context $F, G, E'$ assigns static TDIL to those identifiers that may occur free in $\text{strdec}$. The state $T$ is a set of type names. Rules that generate types (e.g., the rule for datatype declarations) choose type names not in $T$: Types stamped with such names do not share with any types in $B$.

TDEL signature matching complicates the tracking of type names. The rule for opaque signature ascription $\text{strexp} :> \text{sigexp}$ generates types after elaborating $\text{strexp}$ and $\text{sigexp}$. Consider the following structure declaration.

```plaintext
structure A = struct type a = int type b = a end
:> sig type a type b = a end
```

Types $A.a$ and $A.b$ share but neither shares with $\text{int}$. At the level of the TDIL, $A.a$ must be stamped with a new type name and $A.b$ must be stamped with the *same* type name. To handle the book-keeping, a TDIL signature $\Sigma = (T)E$ comprises a set $T$ of bound type names (induced by abstract type specifications) and an environment $E$ describing its components. The example elaborates as follows.

1. The inner structure expression “\text{struct type a = int type b = a end}” elaborates to an environment $E$ mapping type constructors $a$ and $b$ to types stamped with $t_{\text{int}}$ (where $t_{\text{int}}$ is the type name associated with $\text{int}$ in the ambient basis).

2. The signature expression elaborates to a signature $\Sigma = (T)E'$ where $T = \{t\}$ binds one type name and $E'$ maps the type constructors $a$ and $b$ to types stamped with $t$.  

15
uspecs ∈ UnitSpecs = \bigcup_{n \geq 0} \text{UnitSpecs}^n
unitid:Υ_1, \ldots, unitid:Υ_n ∈ \text{UnitSpecs}^n = (\text{UnitId} \times \text{UnitSig})^n
Υ or \begin{aligned}(T)(F, G, E) & \in \text{UnitSig} = \text{TyNameSet} \times (\text{FunEnv} \times \text{SigEnv} \times \text{Env}) \\ Γ or B, U & \in \text{UnitBasis} = \text{Basis} \times \text{UnitEnv} \end{aligned}
U ∈ \text{UnitEnv} = \text{UnitId} \xrightarrow{\text{fin}} \text{Basis}
unitid ∈ \text{UnitId} (\text{unit identifiers})

Figure 10: Static TDIL for unit declaration lists

3. The opaque ascription (a structure expression) elaborates to \( E' \) after the bound type name in \( \Sigma \) is systematically renamed so that it differs from all type names in the ambient basis (e.g., \( t \neq t_{\text{int}} \)).

4. The structure declaration elaborates to an environment \( E_A \) mapping the structure identifier \( A \) to the environment \( E' \) obtained in (3). Thus the types \( A.a \) and \( A.b \) in \( E_A \) are stamped with a fresh type name as required.

The rule for transparent signature ascription \( \text{strexp} : \text{sigexp} \) induces sharing after elaborating \( \text{strexp} \) and \( \text{sigexp} \). Consider the following structure declaration.

```plaintext
structure B = struct type a = int type b = a end
: sig type b end
```

Types \( B.b \) and \( \text{int} \) share. To induce sharing, TD uses capture-avoiding substitution from type names to types: The rule chooses and applies a realization \( \varphi \). This example elaborates as follows.

1. The inner structure expression elaborates as in the preceding example.

2. The signature expression elaborates to a signature \( \Sigma' = (T')E'' \) where \( T' = \{ t' \} \) binds one type name and \( E'' \) maps the type constructor \( b \) to a type stamped with \( t' \).

3. The transparent ascription elaborates to \( \varphi(E'') \) where \( \varphi(\cdot) \) can be applied to any semantic object \( A \) to substitute \( t_{\text{int}} \) (the type name associated with \( b \) in \( E \)) for free occurrences of \( t' \) in \( A \).

4. The structure declaration elaborates to an environment \( E_B \) mapping the structure identifier \( B \) to the environment \( \varphi(E'') \). Thus the type \( B.a \) in \( E_B \) is stamped with \( t_{\text{int}} \) as required.

Both ascription rules employ the signature instantiation and enrichment relations to define TDEL signature matching in terms of TDIL environments and signatures.

**Unit Static Semantics** In Figure 10 we extend the static TDIL for unit declaration lists. These TDIL categories—disjoint from all others—build on TD’s categories TyNameSet, FunEnv, SigEnv, Env, and Basis (see Appendix C). When specifying TDIL we use the following notation.

- \( A \times B \) denotes the cartesian product of \( A \) and \( B \).
- \( A \cup B \) denotes the disjoint union of \( A \) and \( B \).
- \( A \xrightarrow{\text{fin}} B \) denotes the set of partial functions from \( A \) to \( B \) with finite domain.
Judgement...                     Meaning...
\[ \Gamma \vdash udecs : uspecs \]  udecs has specification list uspecs
\[ \Gamma \vdash \text{topdec} : B \]  topdec has basis B
\[ \Gamma \vdash \text{impexp} : F,G,E \]  impexp has components F,G,E

\[ B \vdash \text{intexp} \Rightarrow \Upsilon \]  interface expression elaboration
\[ B \vdash \text{topspec} \Rightarrow F,E \]  top-level specification elaboration
\[ B \vdash \text{funspec} \Rightarrow F \]  functor specification elaboration

\[ \Gamma \vdash \text{udecs} \]  \( \Gamma \) is well-formed
\[ \Gamma \vdash \Upsilon : \text{Sig} \]  \( \Upsilon \) is well-formed
\[ \Gamma \vdash \text{uspecs ok} \]  uspecs is well-formed

\[ \Gamma \vdash E : \Sigma \text{ using } \varphi \]  signature matching
\[ \Sigma \equiv \Sigma' \text{ using } \varphi \]  signature equivalence
\[ T \equiv \Upsilon' \text{ using } \varphi \]  interface equivalence
\[ F,G,E : \Upsilon \text{ using } \varphi \]  interface matching

Figure 11: Unit static semantics

- \( A^n \) denotes a sequence of length \( n \geq 0 \) whose range is a subset of \( A \).

A unit signature (or interface) \( \Upsilon = (T)(F,G,E) \) describes a unit. \( \Upsilon \) specifies the components in environments \( F, G, \) and \( E \), binding type names \( T \) with scope \((F,G,E)\).

A unit specification list \( \text{unitid}_1: \Upsilon_1, \ldots , \text{unitid}_n: \Upsilon_n \) describes a unit declaration list. Writing \( \text{BT}(\Upsilon) \) for the type names bound in \( \Upsilon \), a unit specification list binds \( \text{BT}(\Upsilon_i) \) with scope \( \text{unitid}_{i+1}: \Upsilon_{i+1}, \ldots , \text{unitid}_n: \Upsilon_n \) for each \( 1 \leq i < n \).

A unit basis \( \Gamma = B,U \) (where \( B = T,F,G,E \)) comprises a state \( T \) and a context \( F,G,E,U \). The unit environment \( U \) is a finite map from unit identifiers to bases: If \( U(\text{unitid}) = T',F',G',E' \), then \( T' \subset T \) records the type names generated by \( \text{unitid} \) and the environments \( F',G',E' \) describe its components.

We give a static semantics to unit declaration lists by giving rules for the judgements in Figure 11. The rules appear in Appendix C. These judgements have the following meaning.

- \( \Gamma \vdash udecs : uspecs \). The unit declaration list \( udecs \) matches the unit specification list \( uspecs \). Corresponding unit identifiers must agree and each source unit in \( udecs \) must match its specification in \( uspecs \). For example, the judgement

\[ \Gamma \vdash (\text{unit } \text{unitid} = \text{unit topdec end}, udecs) : (\text{unitid}(T)(F,G,E), uspecs) \]

holds if \( \Gamma \vdash \text{topdec} : T,F,G,E \) and \( \Gamma + T + \{ \text{unitid} \mapsto T,F,G,E \} \vdash udecs : uspecs \).

- \( \Gamma \vdash \text{topdec} : B \). The top-level declaration \( \text{topdec} \) has basis \( B = T,F,G,E \): It generates type names \( T \) and declares the components in \( F,G,E \).

- \( \Gamma \vdash \text{impexp} : F,G,E \). The import expression \( \text{impexp} \) imports the components in \( F,G,E \) from \( \Gamma \).

\( ^5 \)The notation \( \Gamma + T + \{ \text{unitid} \mapsto T,F,G,E \} \) extends the state then the context in \( \Gamma \) (see Appendix C).
• \( B \vdash \text{intexp} \Rightarrow \Upsilon \). The interface expression \( \text{intexp} \) specifies the components in \( \Upsilon \). Since interface expressions do not describe signature declarations, the signature environment \( G \) in \( \Upsilon \) must be empty.

• \( B \vdash \text{topspec} \Rightarrow F, E \). The top-level specification \( \text{topspec} \) specifies the components in \( F, E \). No identifier may be specified twice.

• \( B \vdash \text{funspec} \Rightarrow F \). The functor specification \( \text{funspec} \) specifies the components in \( F \). No identifier may be specified twice.

• \( \Gamma \text{ ok} \). The unit basis \( \Gamma = (T, F, G, E, U) \) is well-formed: \( T \) contains the free type names in \( F, G, E, U \).

• \( \Gamma \vdash \Upsilon : \text{Sig} \). The interface \( \Upsilon = (T)(F, G, E) \) is well-formed: No type name \( t \in T \) occurs in \( \Gamma \)'s state \( T' \) and \( T \cup T' \) contains the free type names in \( F, G, E \).

• \( \Gamma \vdash \text{uspecs ok} \). The unit specification list \( \text{uspecs} \) is well-formed. For example, the judgement

\[ \Gamma \vdash \text{unitid:}\Upsilon, \text{uspecs ok} \]

holds if \( \Gamma \vdash \Upsilon : \text{Sig} \) and \( \Gamma + \text{BT(}\Upsilon) \vdash \text{uspecs ok} \).

• \( E : \Sigma \text{ using } \varphi \). The environment \( E \) matches the signature \( \Sigma \) using the realization \( \varphi \).

• \( \Sigma \equiv \Sigma' \text{ using } \varphi \). The signatures \( \Sigma = (T)E \) and \( \Sigma' = (T')E' \) specify the same components and \( E = \varphi(E') \).

• \( \Upsilon \equiv \Upsilon' \text{ using } \varphi \). The interfaces \( \Upsilon = (T)(F, G, E) \) and \( \Upsilon' = (T')(F', G', E') \) specify the same components and \( F, G, E = \varphi(F', G', E') \).

• \( F, G, E : \Upsilon \text{ using } \varphi \). The environments \( F, G, E \) match the interface \( \Upsilon \) and the realization \( \varphi \) induces the requisite sharing: If the semantic object \( A \) refers to type names bound in \( \Upsilon \), then the semantic object \( \varphi(A) \) refers to corresponding types employed by \( F, G, E \).

### 4.2 Dynamic Semantics

We shall use the TD dynamic semantic judgements given in Figure 9.\(^6\) These judgements have the following meaning.

\[
\begin{array}{ll}
\text{Judgement} & \text{Meaning} \\
\text{s, B \vdash strdec } & \text{structure declaration evaluation} \\
B \vdash \text{fundec } & \text{functor declaration evaluation} \\
IB \vdash \text{sigdec } & \text{signature declaration elaboration} \\
IB \vdash \text{sigexp } & \text{signature elaboration} \\
IB \vdash \text{spec } & \text{specification elaboration} \\
\text{Inter } B = IB & \text{interface basis extraction} \\
E \downarrow I = E' & \text{signature ascription}
\end{array}
\]

Figure 12: TD’s dynamic semantic judgements (summary)
• $s, B \vdash \text{strdec} \Rightarrow E/p, s'$. In state $s$ and context $B$, the structure declaration $\text{strdec}$ evaluates to state $s'$ and either an environment $E$ or an exception packet $p$.

• $B \vdash \text{fundec} \Rightarrow F$. The functor declaration $\text{fundec}$ evaluates to the functor environment $F$. State is not involved: $F$ maps functor identifiers to functor closures.

• $IB \vdash \text{sigdec} \Rightarrow G$. The signature declaration $\text{sigdec}$ declares the signature identifiers in signature environment $G$.

• $IB \vdash \text{sigexp} \Rightarrow I$. The signature expression $\text{sigexp}$ specifies the components in structure interface $I$.

• $IB \vdash \text{spec} \Rightarrow I$. The specification $\text{spec}$ specifies the components in $I$.

• $\text{Inter } B = IB$. The interface basis $IB$ is the value-free part of the basis $B$. (Values are not needed by the signature elaboration judgements.)

• $E \downarrow I = E'$. The environment $E'$ is the environment $E$ cut down to match the structure interface $I$.

The dynamic semantics is defined for mostly type-erased TDEL. Types, type ascriptions, and type qualifications are erased. Signatures are not erased. Both signature ascription and functor application limit the “view” of a structure in case it is opened. Consider the following declarations.

```
structure A = struct val x = 1 val y = 2 end
  : sig val x : int end
val y = 3
open A
```

The value of $y$ is 3 not 2. A related example employs functors:

```
functor F(S : sig val x : int end) =
  struct
    val y = 3
    open S
  end
structure B = F(struct val x = 1 val y = 2 end)
```

The value of $B.y$ is 3 not 2. The dynamic semantics uses structure interfaces to cut down environments when evaluating signature ascriptions and functor applications. To obtain structure interfaces, the dynamic semantics re-elaborates signatures. This “dynamic elaboration” tracks only the status of identifiers, making it simpler than the elaboration performed by the static semantics. For example, it is stateless.

In the evaluation judgement $s, B \vdash \text{strdec} \Rightarrow E/p, s'$, the states $s$ and $s'$ track a set of generated exception names and a memory graph for references. The dynamic semantics accounts for the generativity of exception bindings by stamping exception values with exception names—two TDIL exceptions are equal if they are stamped with the same name. The rules propagate raised exceptions explicitly, referring to them as exception packets $p$. (Compound metavariables like $E/p$ range over the disjoint union of two TDIL categories.)
Figure 13: Unit dynamic semantic objects

**Unit Dynamic Semantics.** In Figure 13 we extend the dynamic TDIL for unit declaration lists. These TDIL categories—disjoint from all others—build on TD’s dynamic TDIL categories Basis and Int (see Appendix D).

As with TDEL signatures, interface expressions are not erased prior to evaluation. An SC import declaration `import unitid : intf` is analogous to a TDEL signature ascription and an open declaration: It limits the “view” of the imported unit. We shall give a dynamic semantics that uses (dynamic) unit interfaces to cut down bases when elaborating SC imports.

A unit interface `UI = FI, I` describes a unit. The structure interface `I` describes structure, type, and value components. The functor interface environment `FI` describes functor components using functor interfaces. A functor interface `I, I'` comprises argument and result structure interfaces. Both are necessary. Consider the following unit declaration list.

```plaintext
unit U1 =
  unit
  functor F(S : sig end) = struct open S val x = 1 val y = 2 end
end,

unit U2 =
  unit
    import U1 :
      intf
        functor F(S : sig val x : int end) : sig val x : int end
      end
    structure A = F(struct val x = 3 end)
    val y = 4
    open A
end
```

The values of `x` and `y` in `U2` are, respectively, 1 and 4 rather than 3 and 2.

A unit basis `Γ = B, U` serves as an evaluation context. The unit environment `U` is a finite map from unit identifiers to bases: If `U(unitid) = F, G, E`, then the environments `F, G, E` record the values obtained by evaluating `unitid`.

We give a dynamic semantics to unit declaration lists by giving rules for the evaluation judgements in Figure 14. The rules appear in Appendix D. These judgements have the following meaning.

- `s, Γ ⊢ udecs ⇒ Γ' / p, s'`. Evaluate the unit declaration list `udecs` to a unit basis `Γ'` or an exception packet `p`.
- `s, Γ ⊢ topdec ⇒ B / p, s'`. Evaluate the top-level declaration `topdec` to a basis `B` or an exception packet `p`.

20
Figure 14: Unit evaluation judgements

- \( \Gamma \vdash \text{implexp} \Rightarrow B \). Evaluate the import expression \( \text{implexp} \) to the basis \( B \).
- \( \text{IB} \vdash \text{topspec} \Rightarrow \text{UI} \). The top-level specification \( \text{topspec} \) specifies the components in the unit interface \( \text{UI} \).
- \( \text{IB} \vdash \text{funspec} \Rightarrow \text{FI} \). The functor specification \( \text{funspec} \) specifies the components in the functor interface environment \( \text{FI} \).

4.3 Linking

We define linking for SMLSC by giving rules for deriving the judgements in Figure 15. A linkset

\( \text{uspecs}_0 \rightarrow \text{exps} \)

comprises imports \( \text{uspecs}_0 \) and exports \( \text{exps} \). Exports may take two forms—a unit declaration list \( \text{udecs} : \text{uspecs} \) or a static TDIL basis \( B \).

- The imports \( \text{uspecs}_0 \) describe the units on which the linkset depends; they must be well-formed in the ambient context and no unit may be described twice. For example, the imports

\( \text{uspecs}_{AB} = A:\text{\( \Upsilon \)}_A, B:\text{\( \Upsilon \)}_B \)

express dependency on units \( A \) and \( B \).

Imports specify assumptions to be satisfied by linking. A linkset with imports \( \text{uspecs}_{AB} \) assumes unit \( B \) declares (at least) the components described by the interface \( \text{\( \Upsilon \)}_B \) but can be linked with (a linkset exporting) a unit \( B \) providing more components.

- The exports \( \text{exps} = \text{udecs} : \text{uspecs} \) are the code associated with the linkset. They may make reference to the linkset’s imports (via free unit identifiers and type names).

Figure 15: Linking judgements
\[ L ::= \text{uspecs} \to \text{exps} \] \text{linkset}
\[ \text{exps} ::= \text{udecs} : \text{uspecs} \] \text{units}
\[ B \] \text{top-level components}

Figure 16: Linkset syntax

- The exports \( \text{exps} = B \) arise during elaboration and record typing information for the top-level declaration associated with the linkset. They may make reference to the linkset’s imports (via free type names).

The dynamic semantics for SMLSC is very simple. The completion judgment \( L \Rightarrow \text{udecs} \) translates a linkset
\[ \cdot \to \text{udecs} : \text{uspecs} \]
with no imports to the unit declaration list \( \text{udecs} \). Under the dynamic semantics for units given in Section 4.2, the resulting unit declaration list evaluates the linkset’s exports from left to right for their side-effects.\(^7\) Evaluation terminates when an uncaught exception is raised or when every export has been evaluated.

We give the full syntax for linksets in Figure 16 and the rules in Appendix E. The remainder of this section explains the rules for linkset merge.

**Notation.** We define the extension of a unit basis by a unit specification list, \( \Gamma + \text{uspecs} \), by\(^8\)
\[
\Gamma + \cdot = \Gamma \\
\Gamma + (\text{unitid}: (T)(F,G,E), \text{uspecs}) = (\Gamma + T + \{\text{unitid} \mapsto T, F, G, E\}) + \text{uspecs}.
\]
We define the type names bound by a unit specification list, \( \text{BT}(\text{uspecs}) \), by
\[
\text{BT}(\text{unitid}_1: \Upsilon_1, \ldots, \text{unitid}_n: \Upsilon_n) = \text{BT}(\Upsilon_1) \cup \cdots \cup \text{BT}(\Upsilon_n).
\]
We define the domain of a unit specification list, \( \text{dom}(\text{uspecs}) \), by
\[
\text{dom}(\text{unitid}_1: \Upsilon_1, \ldots, \text{unitid}_n: \Upsilon_n) = \{\text{unitid}_1, \ldots, \text{unitid}_n\}
\]
and the domain of a linkset’s exports, \( \text{dom}(\text{exps}) \), by
\[
\text{dom}(\text{udecs} : \text{uspecs}) = \text{dom}(\text{uspecs}) \\
\text{dom}(B) = \emptyset.
\]

**Linkset merge.** The rules for linkset merge \( \Gamma \vdash L_1 ++ L_2 \Rightarrow L_3 \) combine \( L_1 \) and \( L_2 \) to produce \( L_3 \). The rules presuppose that \( L_1 \) is well-formed with respect to \( \Gamma \) but permit \( L_2 \) to make reference not only to \( \Gamma \) but to the imports and exports of \( L_1 \).

The rules process the imports in \( L_2 \) from left to right. If \( L_2 \) has no imports, then the following rule applies.
\[
\Gamma \vdash \text{exps} ++ \text{exps}' \Rightarrow \text{exps}''
\]
\[
\Gamma \vdash (\text{uspecs}_0 \to \text{exps}) ++ (\cdot \to \text{exps}') \Rightarrow \text{uspecs}_0 \to \text{exps}''
\]
\(^7\)The unit declaration list \( \text{udecs} \) obtained by completion may be evaluated in the initial state \( s_0 \) and the unit basis \( \Gamma_0 = B_0, \emptyset \), where \( s_0 \) and the initial dynamic basis \( B_0 \) are given in TD [16, Appendix D].
\(^8\)Please see Appendix C for a summary of TDIL notation, including definitions of \( \Gamma + T, \Gamma + U, \) and \( \Gamma(\text{unitid}) \).
L₃ imports what L₁ does. The rules for \( \Gamma \vdash \text{exps} ++ \text{exps}' \Rightarrow \text{exps}'' \) ensure that L₃ exports what L₁ and L₂ do.

Otherwise, the rules examine the first import unitid:Υ in L₂ and distinguish three mutually exclusive cases:

- L₁ exports unitid.

\[
\begin{align*}
\text{unitid} & \in \text{dom}(\text{uspecs}) \\
(\Gamma + \text{uspecs}_0 + \text{uspecs})(\text{unitid}) &= T', F', G', E' \\
F', G', E' : \text{Υ} \text{ using } \varphi \\
L' &:= \varphi(\text{uspecs}_1 \rightarrow \text{udecs}' : \text{uspecs}') \\
\Gamma \vdash (\text{uspecs}_0 \rightarrow \text{udecs} : \text{uspecs}) \Rightarrow L''
\end{align*}
\]

The first premise ensures L₁ exports unitid. The second premise picks out the L₁ export unitid : (T')(F', G', E') for unitid. The third premise matches the exported environments F', G', E' to the imported interface Υ. Linking fails if no match is possible; otherwise, L' is constructed by changing references to the type names bound by Υ in the remainder of L₂ to the types employed by L₁.

- L₁ imports unitid but does not export it.

\[
\begin{align*}
\text{unitid} & \notin \text{dom}(\text{exps}) \\
\text{uspecs}_0 = \text{uspecs}'', \text{unitid}:\text{Υ}', \text{uspecs}''' \\
\text{Υ}' &\equiv \text{Υ} \text{ using } \varphi \\
L' &:= \varphi(\text{uspecs}_1 \rightarrow \text{exps}') \\
\Gamma \vdash (\text{uspecs}_0 \rightarrow \text{exps}) \Rightarrow L''
\end{align*}
\]

The first premise ensures L₁ does not export unitid. The second premise picks out the L₁ import unitid:Υ'; there can be at most once since L₁ is well-formed. Linking fails if Υ and Υ' are not equivalent; otherwise, L' is constructed by changing references to the type names bound by Υ in the remainder of L₂ to the type names bound by Υ'.

- L₁ neither imports nor exports unitid.

\[
\begin{align*}
\text{unitid} & \notin \text{dom}(\text{exps}) \cup \text{dom}(\text{uspecs}_0) \\
\text{Υ}' &\equiv \text{Υ} \text{ using } \varphi \\
\Gamma + \text{BT}(\text{uspecs}_0) &\vdash \text{Υ}' : \text{Sig} \\
L &:= \text{uspecs}_0, \text{unitid}:\text{Υ}' \rightarrow \text{exps} \\
L' &:= \varphi(\text{uspecs}_1 \rightarrow \text{exps}') \\
\Gamma \vdash L \Rightarrow L''
\end{align*}
\]

The first premise ensures that L₁ neither imports nor exports unitid. The next two premises choose an interface Υ' equivalent to Υ but well-formed without reference to the exports of L₁. Linking fails if no such interface exists—when type names exported by L₁ occur in Υ. Otherwise, L is constructed by adding a new import to the imports in L₁ and L' is constructed by changing references to the type names bound in Υ in the remainder of L₂ to the type names bound in Υ'.
4.4 Elaboration

We define a semantics for SMLSC by giving rules for the elaboration judgements in Figure 17. We give the abstract syntax for SMLSC in Figure 18. The elaboration rules appear in Appendix F. These judgements have the following meaning.

- **project ⇒ L**. Elaborate project, using linkset merge to accumulate a resulting linkset L. A source unit is elaborated in a unit basis Γ that declares the imports and exports in L.

- **Γ ⊢ srcunit ⇒ L**. Elaborate the topdec in srcunit to the linkset
  
  \[ udecs_0 \rightarrow T, F, G, E. \]

  The imports udecs₀ arise from the import declarations in topdec. The type names T arise from the types generated by topdec. The environments F, G, and E arise from the declarations in topdec. The result, L, exports a single unit:

  \[ L = udecs_0 \rightarrow srcunit : (T)(F,G,E). \]

- **Γ ⊢ topdec ⇒ L**. Elaborate topdec using linkset merge.

- **Γ ⊢ impexp ⇒ L**. Elaborate impexp using context lookup and intexp elaboration.

5 Implementation

The semantics of SMLSC avoids commitment to the meaning of “compilation,” “linking,” and “completion” to ensure compatibility with various implementation strategies. These phases may be implemented using classical methods (code generation during compilation, object code weaving during linking, and writing an executable for completion), or in other, more novel, ways (such as type checking during compilation, and code generation during linking). The design is, as far as we know, implementable in all current Standard ML compilers without requiring radical changes to their infrastructure.
Parallel Build. A compiler can exploit interfaces to support parallel compilation in order to speed up system build times. A unit can be compiled once interfaces have been inferred for its IC imports. The TILT compiler, which implements an earlier version of the present extension, implements such a strategy. Moreover, it also implements cut-off incremental recompilation [1], where it is able to interrupt the normal cascade of recompilation when a source change does not cause a unit’s interface to change.

Parsing. This presentation of SMLSC provides concrete and abstract syntax, but does not formalize parsing. The only issue that entangles separate compilation and parsing is fixity declarations. To support fixity declarations at parse-time, we include a parsing context in the concrete representation of linksets (object files). A source unit that is incrementally compiled against a linkset is parsed using that linkset’s included parsing context. We do not permit fixity specifications in user-specified interfaces, and therefore they do not affect interface matching or any other part of the semantics.

Note that a library may specify fixity information by placing appropriate declarations in the handoff unit. For example, to describe a matrix library that supplies an infix ** operator for multiplication, we may write the following handoff unit:

```
unit Matrices =
unit
  import MatricesImpl :
    intf
      type matrix
      val ** : matrix * matrix -> matrix
       (* · · · *)
    end
  infix **
end
```

6 Multiple Interfaces for the Same Import

In Section 2 we presented the programming methodology of handoff units. As long as two linksets that import the same unit identifier do so by using the same handoff unit, they will always agree on the interface for that unit and so can be linked together. However, in some situations it may be useful to permit two clients to import the same unit, each with a different interface. Since interface matching, like signature matching, is coercive, this complicates the methodology of definite references by introducing “views” of the same underlying unit.

For example, suppose that two linksets \( L_1 \) and \( L_2 \) import the same unit MathLib at disparate interfaces \( I_1 \) and \( I_2 \). This may happen because the developers of \( L_1 \) and \( L_2 \) compiled using different versions of the handoff unit for MathLib, or because the developers wrote their import interfaces by hand. The link

\[
\text{link}(L_1, L_2)
\]

fails because the linksets are required to agree on the interfaces of their common imports. Aside from recompiling the two linksets to use the same interface, the programmer has several options for resolving this situation. First, she can satisfy the imports by providing the implementation of
The linkset \( L_{\text{glue}} \) imports \texttt{MathLib} at interface \( I \) and then exports it to satisfy the imports in \( L_1 \) and \( L_2 \) at disparate interfaces \( I_1 \) and \( I_2 \).

\texttt{MathLib}:

\[
L'_1 = \text{link}(\texttt{MathLib}, L_1) \\
L = \text{link}(L'_1, L_2)
\]

The first step satisfies the SC import of \texttt{MathLib} in \( L_1 \), as long as the actual interface of \texttt{MathLib} matches the import interface \( I_1 \). The result \( L'_1 \) does not import \texttt{MathLib}, so it does not conflict with the import of \texttt{MathLib} in \( L_2 \). \( L'_1 \) does export \texttt{MathLib}, so if the actual interface of \texttt{MathLib} matches \( I_2 \), then the second link succeeds. Because linking is left-associative, \( L = \text{link}(\texttt{MathLib}, L_1, L_2) \) accomplishes the same thing.

Any implementation of \texttt{MathLib} that satisfies both \( I_1 \) and \( I_2 \) will suffice. Because we do not require unit names to be globally unique, this implementation of \texttt{MathLib} might even import \texttt{MathLib} (again) and then contain some glue code to make it compatible with the two given interfaces \( I_1 \) and \( I_2 \) (Figure 19). We expect such cases to be uncommon, the preferred methodology being to use a single handoff unit for all clients.

7 Related Work

There are several closely related systems that influenced the design of SMLSC.

The notion of linkset in SMLSC comes from Cardelli’s investigation of separate compilation and type-safe linking in the simply-typed \( \lambda \)-calculus [5]. Our formalization of linking extends these ideas to support the Standard ML module system including signature subtyping, abstract types, and module and type definitions in structures.

Harper and Pierce [11] discuss language design for module systems, including separate compilation. Particularly relevant to the current work is their discussion of sharing of abstract types. They describe the use of definite references to avoid the coherence problems (and excess sharing specifications) that arise from aliasing.

The notion of a handoff unit bears some resemblance to the use of .h files in C. The presence of function prototypes in a .h file provides an interface for application code that includes that header file. Code that references a prototyped function triggers a link-time demand for that function. The degree of link-time type-checking varies across C implementations. Usually, type correctness is assured by programming conventions.

Glew and Morrisett [8] describe separate compilation for Typed Assembly Language [19]. Their language, MTAL, permits type definitions, abstract types, and polymorphic types in interfaces and supports recursive linking.

Jim [14] describes a \( \lambda \)-calculus \( P_2 \) with rank 2 intersection types that has principal typings. The principal typings property means that from a term \( M \), one can infer both \( \Gamma \) and \( \tau \) such that any typing derivation \( \Gamma' \vdash M : \tau' \) is an instance of \( \Gamma \vdash M : \tau \). In a system with principal typings, program fragments can be separately compiled without context information, meaning that SC imports need not even specify interfaces. Standard ML, however, does not have principal typings.
It remains an open problem to design a type system that supports principal types for features such as abstract and recursive types, and type definitions in modules.

**Objective Caml.** The separate compilation system of Objective Caml (O'Caml) [15] is similar in many regards to SMLSC. The declaration of a unit $U$ is an O'Caml module stored within a file called $U.ml$. The interface for $U$ may optionally be given in a file called $U.mli$. If the interface is present, other units depending on $U$ can compile even if the implementation is not available, just as in SMLSC. Because the filename of an interface indicates the unit that it describes, O'Caml interfaces play the role of handoff units in SMLSC. Additionally, O'Caml’s use of the filesystem to provide a canonical location for each unit and interface means that all unit references are definite.

On the other hand, O'Caml’s dependence on the filesystem means that the language is not independent from its environment. For instance, unit names are limited to valid filenames on the host system, and restructuring a project on disk may force changes to the code. Another significant difference is that O'Caml conflates the notions of units and modules. This earns O'Caml some conceptual economy, but it makes it impossible to separate the notions of top-level declarations and structure components. This makes it necessary to support signature and functor definitions within structures, so such a choice would not be compatible with our design principle of conservativity over Standard ML. Finally, unlike SMLSC, O'Caml and its separate compilation system are defined informally in terms of their implementation.

**Moscow ML.** The Moscow ML [20] compiler for Standard ML supports a separate compilation system nearly identical to Objective Caml’s. Moscow ML extends the Standard ML module system to allow (among other things) functor and signature declarations in structures and specifications for them in signatures. Then, like O'Caml, units are structures. In contrast, SMLSC does not require any changes to the Standard ML module language.

Other Standard ML implementations include mechanisms for breaking programs up into compilation units. None support separate compilation in the sense we use it here; they use the term to mean cut-off incremental recompilation (recall Section 5).

**SML/NJ CM.** The Compilation Manager for Standard ML of New Jersey (CM) [3] is a tool for compiling Standard ML programs spread across many source files. CM permits a program to be divided into a hierarchy of libraries [4]. A library comprises a list of imported libraries, Standard ML source files, and a list of symbols exported by the library. Dependencies between libraries are explicit but dependencies among the source files in a library are inferred [2, 9]. CM provides control over the identifiers visible to a source file, and supports conditional compilation, parallel compilation, and cut-off incremental recompilation. CM provides no way for the programmer to write interfaces nor to compile against unimplemented units. SMLSC is not a replacement for CM; we believe that dependency analysis and recompilation tools are useful, and that SMLSC provides a good linguistic target for such tools.

**ML Basis.** The MLton compiler [18] and ML Kit [17] implement a language called ML Basis. A “basis” in their terminology is what we call a unit. An ML Basis program is a series of declarations, including a binding construct for bases and an open construct for basis identifiers. These are analogous to SMLSC’s unit declaration and IC import declaration. Like SMLSC, the order of compilation entities is explicit, and thus each program has unambiguous meaning. ML Basis is given a formal semantics [6] in terms of The Definition of Standard ML. The implementation
of ML Basis in the ML Kit supports cut-off incremental recompilation based on Elsman’s thesis work [7]. Like CM, ML Basis does not provide a way for programmers to write down interfaces or separately compile against unimplemented bases.

8 Conclusion

We have presented an extension to Standard ML for separate compilation called SMLSC. Its focus is the unit, a program fragment that can depend on other program fragments through either separate or incremental compilation. Via the programming idiom of handoff units—that uses both separate and incremental compilation—we limit the number and complexity of linguistic mechanisms while supporting a convenient programming style. Our formal and abstract definition of the language ensures that it is unambiguously specified, and admits a variety of implementation strategies.

Acknowledgements

We thank Carsten Varming for comments on a draft of this manuscript.

References


## A TS Linking Rules

The typed semantics defines a closed structure $\text{mod}_{\text{basis}};\text{sig}_{\text{basis}}$ serving as an initial basis for the TSIL. The elaborator assumes $\Gamma$ declares $\text{basis};\text{sig}_{\text{basis}}$, which includes components such as the built-in Match exception. This basis structure is introduced in Rule 6 for completion and Rule 12 for elaboration of source units in projects.

We use the following definitions and notation.

- Writing $\text{BV}(\text{dec})$ for the variable declared by $\text{dec}$, we define the bound variables of a structure declaration list, $\text{BV}(\text{sdecs})$, by

$$\text{BV}(\text{lab}_1 \triangleright \text{dec}_1, \ldots, \text{lab}_n \triangleright \text{dec}_n) = \{\text{BV}(\text{dec}_1), \ldots, \text{BV}(\text{dec}_n)\}.$$  

A linkset

$$L = \text{sdecs}_0 \rightarrow \text{sbnds} : \text{sdecs}; S$$
binds variables \( \text{BV}(sdecs_0) \) with scope \( \text{sbnds} : sdecs \); and variables \( \text{BV}(sdecs) \) with scope \( S \). We write \( \text{BV}(L) \) for \( \text{BV}(sdecs_0) \cup \text{BV}(sdecs) \).

- For readability, we sometimes elide variables in structure bindings and declarations. It should be immediately obvious how to consistently restore these with fresh variables.
- We assume that unit identifiers are disjoint from all other identifier classes.
- We assume that the TS overbar injection \( \overline{\cdot} \) maps identifiers of different classes to different labels and that there are infinitely many labels not in its range.
  We assume that its range includes neither the distinguished label \( \text{basis} \), nor the labels chosen fresh in the rules.
- Structure declaration lists \( sdecs \), signature abbreviations \( S \), and so on specify lists of elements. We adopt the following notation for lists.
  - We denote by \((\cdot, \cdot)\) the operation of syntactic concatenation; for example, \( S, S' \).
  - We sometimes use pattern matching at the left end of a list, writing \( \text{sigid} = \text{sig}, S \) to match the first binding in the list.
  - We usually omit the initial \( : \); for example,
    \[
    \text{sigid}_1 = \text{sig}_1, \ldots, \text{sigid}_1 = \text{sig}_1.
    \]
- We define the domain of a signature abbreviation, \( \text{dom}(S) \), by
  \[
  \begin{align*}
  \text{dom}(\cdot) &= \emptyset \\
  \text{dom}(S, \text{sigid} = \text{sig}) &= \text{dom}(S) \cup \{\text{sigid}\} \\
  \text{dom}(S, \text{unitid} = S_u) &= \text{dom}(S) \cup \{\text{unitid}\}.
  \end{align*}
  \]
- We define the function \( S \mathbin{+\!+} S' \) by
  \[
  \begin{align*}
  (\cdot \mathbin{+\!+} S') &= S' \\
  ((\text{sigid} = \text{sig}, S) \mathbin{+\!+} S') &= \\
  &\begin{cases} 
  \text{sigid} = \text{sig}, S'' \text{ if } \text{sigid} \notin \text{dom}(S'') \\
  S'' \text{ otherwise}
  \end{cases} \\
  \text{where } S'' = S \mathbin{+\!+} S' \\
  ((\text{unitid} = S_u, S) \mathbin{+\!+} S') &= \\
  &\begin{cases} 
  \text{unitid} = S_u, S'' \text{ if } \text{unitid} \notin \text{dom}(S'') \\
  S'' \text{ otherwise}
  \end{cases} \\
  \text{where } S'' = S \mathbin{+\!+} S'.
  \end{align*}
  \]
  It concatenates \( S \) and \( S' \), making the result well-formed by dropping signature abbreviations if \( \text{dom}(S) \cap \text{dom}(S') \neq \emptyset \).
- We write both “\(=\)” and “\(::=\)” in side-conditions. Interpreting the rules algorithmically, the former pattern-matches inputs, and the latter specifies an output.
\[ \text{decs} \vdash L \text{ ok} \]

\[ \begin{align*}
\text{decs} & \vdash sdecs_0 \text{ ok} \\
\text{decs}, sdecs_0 & \vdash \text{sbnds : sdecs} \\
\text{decs}, sdecs_0, sdecs & \vdash S \text{ ok} \\
\text{sbnds} & = \text{lab}_1 \triangleright \text{var}_1 : [sdecs_1], \ldots, \text{lab}_n \triangleright \text{var}_n : [sdecs_n] \\
\text{decs} & \vdash sdecs_0 \rightarrow \text{sbnds : sdecs} ; S \text{ ok}
\end{align*} \]

\textbf{Rule 1:} Imports are restricted to structures. The elaborator in Appendix B needs nothing else.

\[ \text{decs} \vdash S \text{ ok} \]

\[ \begin{align*}
\vdash & \text{decs ok} \\
\text{decs} & \vdash \cdot \text{ ok}
\end{align*} \]

\[ \begin{align*}
\text{decs} & \vdash \text{sig : Sig} \\
\text{decs} & \vdash S \text{ ok} \\
\text{sigid} & \not\in \text{dom}(S) \\
\text{decs} & \vdash \text{sigid} = \text{sig}, S \text{ ok}
\end{align*} \]

\[ \begin{align*}
\text{decs} & \vdash S' \text{ ok} \\
\text{decs} & \vdash S \text{ ok} \\
\text{unitid} & \not\in \text{dom}(S) \\
S' & = (\text{sigid}_1 = \text{sig}_1, \ldots, \text{sigid}_n = \text{sig}_n) \\
\text{decs} & \vdash \text{unitid} = S', S \text{ ok}
\end{align*} \]

\[ L \rightsquigarrow \text{exp} : \{\} \]

\[ \begin{align*}
\text{lab} & \not\in \text{dom}(sdecs) \\
\cdot & \rightarrow \text{sbnds : sdecs} ; S \rightsquigarrow [\text{sbnds}, \text{lab} = \{\}, \text{lab} : \{\}]
\end{align*} \]

\[ L_{\text{basis}} := \cdot \rightarrow \text{mod}_{\text{basis}} : \text{basis} = \text{sig}_{\text{basis}}. \cdot \\
\vdash L_{\text{basis}} + + L \rightsquigarrow L' \quad L' \rightsquigarrow \text{exp} : \{\} \\
L \rightsquigarrow \text{exp} : \{\} \]

\[ \text{decs} \vdash L + + L' \rightsquigarrow L'' \]

\[ L = sdecs_0 \rightarrow \text{sbnds : sdecs} ; S \\
\text{decs} \vdash L + + (\cdot \rightarrow \text{sbnds'} : sdecs' ; S') \rightsquigarrow \\
\text{sdecs}_0 \rightarrow \text{sbnds} + + \text{sbnds'} : \text{sdecs} + + \text{sdecs'} ; S + + S' \\
\text{sdecs} = \text{sdecs}'' ; \text{lab} \triangleright \text{var} \triangleright \text{sig} ; \text{sdecs}''' \]

\[ \text{decs}, \text{sdecs}_0, \text{sdecs} \vdash \text{sub var} \triangleright \text{sig} \leftarrow \text{mod} : \text{sig}'' \\
\text{sbnd} := 1 \triangleright \text{var} = \text{mod} \\
\text{sdec} := 1 \triangleright \text{var} = \text{sig}'' \\
L := \text{sdecs}_0 \rightarrow \text{sbnds} + + \text{sbnd} : \text{sdecs} + + \text{sdec} ; S \\
\text{decs} \vdash L + + (\text{sdecs}_1 \rightarrow \text{sbnds'} : \text{sdecs'} ; S') \rightsquigarrow L'' \\
\text{decs} \vdash (\text{sdecs}_0 \rightarrow \text{sbnds : sdecs} ; S) + + \\
(\text{lab} \triangleright \text{var} = \text{sig}, \text{sdecs}_1 \rightarrow \text{sbnds'} : \text{sdecs'} ; S') \rightsquigarrow L'' \]

31
\[
\begin{align*}
lab \not\in \text{dom}(\text{sdecs}) \\
\text{sdecs}_0 &= \text{sdecs}''', \ lab \triangleright \text{var}': \text{sig}', \ \text{sdecs}''' \\
\text{decs}, \ \text{sdecs}_0, \ \text{sdecs} &\vdash \text{sig} \equiv \text{sig}' : \text{Sig} \\
L' &:= \{\text{var}'/\text{var}\}(\text{decs}_1 \rightarrow \text{sbnds}' : \text{sdecs}'; S') \\
\text{decs} &\vdash (\text{sdecs}_0 \rightarrow \text{sbnds} : \text{sdecs}; S)[++L'] \rightsquigarrow L''
\end{align*}
\]

\[
\begin{align*}
\text{decs} &\vdash (\text{sdecs}_0 \rightarrow \text{sbnds} : \text{sdecs}; S)[++] \\
(lab \triangleright \text{var} : \text{sig}, \text{decs}_1 \rightarrow \text{sbnds}' : \text{sdecs}'; S') &\rightsquigarrow L''
\end{align*}
\]

\[
\begin{align*}
lab \not\in \text{dom}(\text{sdecs}) \cup \text{dom}(\text{sdecs}_0) \\
\text{decs}, \ \text{sdecs}_0, \ \text{sdecs} &\vdash \text{sig} \equiv \text{sig}' : \text{Sig} \\
\text{decs}, \ \text{sdecs}_0 &\vdash \text{sig}' : \text{Sig} \\
L &:= \text{sdecs}_0, \ lab \triangleright \text{var} : \text{sig}' \rightarrow \text{sbnds} : \text{sdecs}; S \\
\text{decs} &\vdash L[++(\text{decs}_1 \rightarrow \text{sbnds}' : \text{sdecs}'; S')] \rightsquigarrow L''
\end{align*}
\]

\[
\begin{align*}
\text{decs} &\vdash (\text{sdecs}_0 \rightarrow \text{sbnds} : \text{sdecs}; S)[++] \\
(lab \triangleright \text{var} : \text{sig}, \text{decs}_1 \rightarrow \text{sbnds}' : \text{sdecs}'; S') &\rightsquigarrow L''
\end{align*}
\]
B TS Elaboration Rules

We change the TS elaborator to expand signature abbreviations. First, we modify every TS elaboration judgement and rule using a TS elaboration context $sdecs$ to use $sdecs; S$. A context $sdecs; S$ binds variables $\text{BV}(sdecs)$ with scope $S$. We define $\text{BV}(\Gamma)$ by $\text{BV}(sdecs)$. Second, we extend the syntax for TSEL signature expressions:

$$\text{sigexp ::= ... sigid \hspace{1cm} signature identifier}$$

Finally, we extend the TS judgement $\Gamma \vdash \text{sigexp} \rightsquigarrow \text{sig} : \text{Sig}$, adding the rule

$$\begin{align*}
\Gamma \vdash \text{ctx sigid} \rightsquigarrow \text{sig} : \text{Sig} \\
\Gamma \vdash \text{sigid} \rightsquigarrow \text{sig} : \text{Sig}
\end{align*}$$

to elaborate signature identifiers.

We use the following definitions and notation.

- To extend an elaboration context $\Gamma = sdecs; S$, we write
  
  $\Gamma, \text{dec for } sdecs, 1 \triangleright \text{dec}; S$,
  
  $\Gamma, sdecs' \text{ for } sdecs, sdecs'; S$, and
  
  $\Gamma, S' \text{ for } sdecs; S, S'$.

  We also define a function $R(sdecs)$ that renames the labels in $sdecs$ to make them inaccessible to identifier resolution:

  $$R(\text{lab}_1 \triangleright \text{dec}_1, \ldots, \text{lab}_n \triangleright \text{dec}_n) = 1 \triangleright \text{dec}_1, \ldots, 1 \triangleright \text{dec}_n.$$  

- We define a function $U(sdecs)$ that drops the labels in $sdecs$:

  $$U(\text{lab}_1 \triangleright \text{dec}_1, \ldots, \text{lab}_n \triangleright \text{dec}_n) = \text{dec}_1, \ldots, \text{dec}_n.$$  

- When an elaboration context $\Gamma = sdecs; S$ appears in a judgement requiring an IL context $\text{decs}$, we implicitly coerce $\Gamma$ to $U(sdecs)$.

- We define the substitution function $\sigma(\text{var}, sdecs, S)$ by

  $$\begin{align*}
  \sigma(\text{var}, \cdot, S) &= S \\
  \sigma(\text{var}, (\text{lab} \triangleright \text{dec}, sdecs), S) &= \\
  &\{\text{var}.\text{lab}/\text{BV}(\text{dec})\}\sigma(\text{var}, sdecs, S)
  \end{align*}$$

  where $\{\text{path}/\text{var}\}S$ denotes the capture-free substitution of $\text{path}$ for free occurrences of $\text{var}$ in $S$. Rule 14 uses $\sigma$ to elaborate source units.

$$\text{project } \rightsquigarrow \text{L}$$

$$\cdot \rightsquigarrow (\cdot \rightarrow \cdot : \cdot)$$  

(11)
\[
\begin{align*}
\text{project} & \rightsquigarrow L \quad \text{basis} \notin \text{BV}(L) \\
L &= \text{sdecs}_0 \rightarrow \text{sbnds} : \text{sdecs}; S \\
\Gamma &:= \text{basis} \cdot \text{sig}_\text{basis}, R(\text{sdecs}_0), \text{sdecs}; S \\
\Gamma &\vdash \text{srcunit} \rightsquigarrow L' \quad \text{var} \notin \text{BV}(L') \\
s\text{decs}'_1 \rightarrow \text{sbnds}' : \text{sdecs}' ; S' := \{\text{var} / \text{basis}\} L' \\
s\text{decs}_1 &:= \text{basis} \cdot \text{var} : \text{sig}_\text{basis}, s\text{decs}'_1 \\
\Gamma &\vdash L + + (s\text{decs}_1 \rightarrow \text{sbnds}' : s\text{decs}' ; S') \rightsquigarrow L''
\end{align*}
\]

Rule 12: The side-condition \(\text{basis} \notin \text{BV}(L)\) can always be achieved by renaming bound variables in \(L\).

\[
\begin{align*}
\text{project} & \rightsquigarrow L \\
\text{BV}(L) \cap \text{BV}(L') &= \emptyset \\
\vdash L' \ \text{ok} \\
\vdash L + + L' \rightsquigarrow L''
\end{align*}
\]

(13)

\[
\Gamma \vdash \text{srcunit} \rightsquigarrow L
\]

\[
\begin{align*}
\Gamma &\vdash \text{topdec} \rightsquigarrow L \\
L &= \text{sdecs}_0 \rightarrow \text{sbnds} : \text{sdecs}; S \\
\text{var} &\notin \text{BV}(\Gamma) \cup \text{BV}(L) \\
\text{sbnds}' &:= \text{unitid} \cdot \text{var} = [\text{sbnds}] \\
\text{sdecs}' &:= \text{unitid} \cdot \text{var} = [\text{sdecs}] \\
S' &:= \text{unitid} = \sigma(\text{var}, \text{sdecs}, S)
\end{align*}
\]

(14)

\[
\Gamma \vdash \text{unit} \ \text{unitid} = \text{topdec} \rightsquigarrow \\
\text{sdecs}_0 \rightarrow \text{sbnds}' : \text{sdecs}' ; S'
\]

\[
\Gamma \vdash \text{topdec} \rightsquigarrow L
\]

\[
\begin{align*}
\Gamma &\vdash \text{impexp} \rightsquigarrow L \\
\Gamma &\vdash \text{import} \ \text{impexp} \rightsquigarrow L
\end{align*}
\]

(15)

\[
\begin{align*}
\Gamma &\vdash \text{strdec} \rightsquigarrow \text{sbnds} : \text{sdecs} \\
\Gamma &\vdash \text{strdec} \rightsquigarrow \cdot \rightarrow \text{sbnds} : \text{sdecs} ; \cdot
\end{align*}
\]

(16)

\[
\begin{align*}
\Gamma &\vdash \text{sigbind} \rightsquigarrow S \\
\Gamma &\vdash \text{signature} \ \text{sigbind} \rightsquigarrow \cdot \rightarrow \cdot ; S
\end{align*}
\]

(17)
Γ ⊢ topdec ⇔ sdecs0 → sbnds : sdecs; S
var /∈ BV(Γ) ∪ BV(sdecs0)
Γ, R(sdecs0), 1* ▷ var:[sdecs], S ⊢ topdec' ⇔ L'
L := sdecs0 → 1* ▷ var = [sbnds] : 1* ▷ var:[sdecs];
Γ ⊢ L++L' ⇔ L''

(18)

Γ ⊢ local topdec in topdec' end ⇔ L''
Γ ⊢ topdec ⇔ L
L = sdecs0 → sbnds : sdecs; S
Γ, R(sdecs0), sdecs, S ⊢ topdec' ⇔ L'
Γ ⊢ L++L' ⇔ L''

(19)

Γ ⊢ impexp ⇔ L

Γ ⊢ ctx unitid ⇔ var : sig
Γ ⊢ ctx unitid ⇔ S var' /∈ BV(Γ)
L := unitid ▷ var': sdecs → 1* ▷ var' : 1*: sig;
S
Γ ⊢ unitid ⇔ L

(20)

Rule 20: Rules 12, 18, and 19 use R(·) to hide imported units from IR imports. The signature sig should be fully selfified.

Γ ⊢ spec ⇔ sdecs var' /∈ BV(Γ)
Γ, var' : [sdecs] ⊢ var' : sig
L := unitid ▷ var': sdecs → 1* ▷ var' : 1*: sig;
Γ ⊢ unitid : intf spec end ⇔ L

(21)

Rule 21: The signature sig should be fully selfified.

Γ ⊢ impexp ⇔ L  Γ ⊢ impexp' ⇔ L'
BV(L) ∩ BV(L') = ∅  Γ ⊢ L++L' ⇔ L''

(22)

Γ ⊢ sigbind ⇔ S

Γ ⊢ sigexp ⇔ sig : Sig S := sigid=sig
(Γ ⊢ sigbind ⇔ S' sigid /∈ dom(S'))
Γ ⊢ sigid = sigexp (and sigbind) ⇔ S', S'

(23)

Rule 23: Either all optional elements or none must be present.
\[ \begin{align*}
\Gamma \vdash_{ctx} \text{sigid} & \rightsquigarrow \text{sig} : \text{Sig} \\
sdecs; S, \text{sigid}=\text{sig} \vdash \text{sigid} & \rightsquigarrow \text{sig} : \text{Sig} \quad (24) \\
\text{sigid}' \neq \text{sigid} \\
sdecs; S \vdash \text{sigid} & \rightsquigarrow \text{sig} : \text{Sig} \quad (25) \\
sdecs; S, \text{sigid}'=\text{sig}' \vdash \text{sigid} & \rightsquigarrow \text{sig} : \text{Sig} \quad (26) \\
sdecs; S \vdash \text{sigid} & \rightsquigarrow \text{sig} : \text{Sig} \\
sdecs; S, \text{unitid}=S' \vdash \text{sigid} & \rightsquigarrow \text{sig} : \text{Sig} \quad (27) \\
sdecs; S, \text{unitid}=S' \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
\text{unitid}' \neq \text{unitid} \\
sdecs; S \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
sdecs; S, \text{unitid}'=S' \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
sdecs; S \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
sdecs; S, \text{sigid}=\text{sig} \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
\Gamma \vdash_{ctx} \text{unitid} & \rightsquigarrow \text{S} \\
sdecs; S, \text{unitid}=S' \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
\text{unitid}' \neq \text{unitid} \\
sdecs; S \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
sdecs; S, \text{unitid}'=S' \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
sdecs; S \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
sdecs; S, \text{unitid}=S' \vdash \text{unitid} & \rightsquigarrow \text{S'} \\
\Gamma \; \text{ok} \\
\vdash U(sdecs) \; \text{ok} \\
sdecs; \cdot \; \text{ok} \quad (30) \\
sdecs; S \; \text{ok} \quad sdecs \vdash \text{sig} : \text{Sig} \\
sdecs; S, \text{sigid}=\text{sig} \; \text{ok} \quad (31) \\
sdecs; S \; \text{ok} \quad sdecs \vdash S' \; \text{ok} \\
S' = (\text{sigid}_1=\text{sig}_1, \ldots, \text{sigid}_n=\text{sig}_n) \\
sdecs; S, \text{unitid}=S' \; \text{ok} \quad (32)
\end{align*} \]

36
\[ B \text{ or } T, F, G, E \in \text{Basis} = \text{TyNameSet} \times \text{FunEnv} \times \text{SigEnv} \times \text{Env} \]
\[ T \in \text{TyNameSet} = \text{Fin(TyName)} \]
\[ F \in \text{FunEnv} = \text{FunId} \overset{\text{fin}}{\rightarrow} \text{FunSig} \]
\[ G \in \text{SigEnv} = \text{SigId} \overset{\text{fin}}{\rightarrow} \text{Sig} \]
\[ E \text{ or } \langle SE, TE, VE \rangle \in \text{Env} = \text{StrEnv} \times \text{TyEnv} \times \text{ValEnv} \]
\[ \Phi \text{ or } \langle T \rangle \langle E, \langle T' \rangle \rangle \in \text{FunSig} = \text{TyNameSet} \times (\text{Env} \times \text{Sig}) \]
\[ \Sigma \text{ or } \langle T \rangle \langle E \rangle \in \text{Sig} = \text{TyNameSet} \times \text{Env} \]
\[ SE \in \text{StrEnv} = \text{StrId} \overset{\text{fin}}{\rightarrow} \text{Env} \]
\[ TE \in \text{TyEnv} = \text{TyCon} \overset{\text{fin}}{\rightarrow} \text{TyStr} \]
\[ VE \in \text{ValEnv} = \text{VId} \overset{\text{fin}}{\rightarrow} \text{TypeScheme} \times \text{IdStatus} \]
\[ t \in \text{TyName} \text{ (type names)} \]
\[ \text{funid} \in \text{FunId} \text{ (functor identifiers)} \]
\[ \text{sigid} \in \text{SigId} \text{ (signature identifiers)} \]
\[ \text{strid} \in \text{StrId} \text{ (structure identifiers)} \]
\[ \text{tycon} \in \text{TyCon} \text{ (type constructors)} \]
\[ \text{vid} \in \text{VId} \text{ (value identifiers)} \]

Figure 20: Static TDIL for Standard ML (summary). \text{Fin}(A) \text{ denotes the set of finite subsets of } A

C Unit Static Semantic Rules

We use the following definitions and notation.

- **TD’s static semantics.** We recall the static TDIL for Standard ML in Figure 20.
  - We write \( E_\emptyset \) for the empty environment (\( \{\} \), \( \{\} \), \( \{\} \)).
  - We write tynames \( A \) for the set of free type names in the semantic object \( A \).
  - We write tyvars \( A \) for the set of free type variables in the semantic object \( A \) [16, Section 4.2].
  - For any semantic object \( A \), we write \( A \text{ wf} \) for “every type structure occurring in \( A \) is well-formed” [16, Section 4.9].
  - We assume familiarity with realizations \( \varphi \) and their support \( \text{Supp} \varphi \) [16, Section 5.2].
  - We assume familiarity with signature instantiation and enrichment [16, Sections 5.3 and 5.5].

- **Sets.** We write \( A \setminus B \) for \( \{ a \in A \mid a \notin B \} \).
  - We write \( \# A \) for the cardinality of the finite set \( A \).

- **Finite maps.** The domain of a finite map \( f : A \overset{\text{fin}}{\rightarrow} B \) is the set \( \text{dom}(f) \) of those \( a \in A \) for which \( f \) is defined.
  - Finite maps may be specified explicitly; for example \( \{ a_1 \mapsto b_1, \ldots, a_n \mapsto b_n \} \) \( (n \geq 0) \).
  - We extend this notation to a form of set comprehension, writing \( \{ a \mapsto b \mid \phi \} \) for the map taking every \( a \) satisfying \( \phi \) to \( b(a) \).

- **Extension.** To extend finite maps \( f, g : A \overset{\text{fin}}{\rightarrow} B \), we define \( f + g : A \overset{\text{fin}}{\rightarrow} B \) by
  \[
  (f + g)(a) = \begin{cases} 
  g(a) & \text{if } a \in \text{dom}(g) \\
  f(a) & \text{otherwise.}
  \end{cases}
  \]
We define $T + T'$ by $T \cup T'$.

We extend semantic objects componentwise; for example, $E + E'$ and $B + B'$ have the evident definitions.

We extend components in a semantic object when it is unambiguous to do so; for example,

$$\Gamma + U = B, U' + U$$
$$\Gamma + B' = B + B', U'$$
$$\Gamma + T = B + T, U' = (T' + T, F, G, E), U'$$

where $\Gamma = B, U'$ and $B = T', F, G, E$.

We lift application to finite maps in semantic objects when it is unambiguous to do so; for example,

$$\Gamma(\text{unitid}) = U(\text{unitid})$$
$$B(\text{strid}) = E(\text{strid}) = SE(\text{strid})$$

where $\Gamma = B, U$; $B = T, F, G, E$; and $E = (SE, TE, VE)$.

- **Projection.** We write $(\cdot \text{ of } \cdot)$ for projection from semantic objects; for example,

$$T \text{ of } \Gamma = T \text{ of } B = T'$$

where $\Gamma = B, U$ and $B = T', F, G, E$.

\[
\Gamma \vdash \text{udecs : uspecs}
\]

\[
\begin{array}{c}
\Gamma \text{ ok} \\
\hline
\Gamma \vdash \cdot : \cdot
\end{array}
\] (33)

\[
\begin{array}{c}
\text{srcunit} = (\text{unit unitid = unit topdec end}) \\
\Upsilon = (T)(F, G, E)
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \text{topdec} : T, F, G, E \\
\Gamma + T + \{\text{unitid $\mapsto$ } T, F, G, E\} \vdash \text{udecs : uspecs}
\end{array}
\] (34)

\[
\begin{array}{c}
\Gamma \vdash (\text{srcunit, udecs}) : (\text{unitid: $\Upsilon$, uspecs})
\end{array}
\]

\[
\Gamma \vdash \text{topdec} : B
\]

\[
\begin{array}{c}
\Gamma \vdash \text{impexp} : F, G, E \\
\hline
\Gamma \vdash \text{import} \text{ impexp} : \emptyset, F, G, E
\end{array}
\] (35)

\[
\begin{array}{c}
B \vdash \text{strdec } \Rightarrow E \\
T := \text{tynames } E \setminus (T \text{ of } B) \\
\text{tyvars } E = \emptyset
\end{array}
\]

\[
\begin{array}{c}
B, U \vdash \text{strdec} : T, \{\}, \{\}, E
\end{array}
\] (36)

\[
\begin{array}{c}
B \vdash \text{sigdec } \Rightarrow G
\end{array}
\]

\[
\begin{array}{c}
B, U \vdash \text{sigdec} : \emptyset, \{\}, G, E_\emptyset
\end{array}
\] (37)

Rule 37: TD’s Rules 65 and 67 ensure that tynames $G = \emptyset$. TD’s rules for $B \vdash \text{spec } \Rightarrow E$ ensure that tyvars $G = \emptyset$. 

38
\[ B \vdash \text{fundec} \Rightarrow F \quad T := \text{tynames } F \setminus (T \text{ of } B) \quad \text{tyvars } F = \emptyset \]
\[ B, U \vdash \text{fundec : } T, F, \{\}, E_\emptyset \]
\[ \Gamma \vdash \text{topdec : } B \quad \Gamma + B \vdash \text{topdec' : } T', F', G', E' \quad T := (T \text{ of } B) \cup T' \]
\[ \Gamma \vdash \text{local topdec in topdec' end : } T, F', G', E' \]
\[ \Gamma \vdash \text{topdec : } B \quad \Gamma + B \vdash \text{topdec' : } B' \]
\[ \Gamma \vdash \text{topdc topdec' : } B + B' \]

\[ \Gamma \vdash \text{imexp : } F, G, E \]
\[ \Gamma(\text{unitid}) = T, F, G, E \]
\[ \Gamma \vdash \text{unitid : } F, G, E \]
\[ B \text{ of } \Gamma \vdash \text{intexp} \Rightarrow \Upsilon \quad \Upsilon = (T)(F, \{\}, E) \]
\[ \Gamma(\text{unitid}) = T', F', G', E' \quad F', G', E' : \Upsilon \text{ using } \varphi \]
\[ \Gamma \vdash (\text{unitid : } \text{intexp}) : \varphi(F), \{\}, \varphi(E) \]
\[ \Gamma \vdash \text{imexp : } F, G, E \quad \Gamma \vdash \text{imexp' : } F', G', E' \]
\[ \Gamma \vdash \text{imexp imexp' : } F+F', G+G', E+E' \]

\[ B \vdash \text{intexp} \Rightarrow \Upsilon \]
\[ B \vdash \text{topspec} \Rightarrow F, E \quad T := (\text{tynames } F, E) \setminus (T \text{ of } B) \]
\[ B \vdash \text{intf topspec end} \Rightarrow (T)(F, \{\}, E) \]

\[ B \vdash \text{topspec} \Rightarrow F, E \]
\[ B \vdash \text{spec} \Rightarrow E \]
\[ B \vdash \text{spec} \Rightarrow \{\}, E \]
\[ B \vdash \text{funcspec} \Rightarrow F \]
\[ B \vdash \text{functor funspec} \Rightarrow F, \{\} \]
\[ B \vdash \text{topspec} \Rightarrow F, E \quad B + E \vdash \text{topspec' } \Rightarrow F', E' \]
\[ \text{dom}(F) \cap \text{dom}(F') = \emptyset \quad \text{dom}(E) \cap \text{dom}(E') = \emptyset \]
\[ B \vdash \text{topspec topspec' : } F+F', E+E' \]
$B \vdash \text{funspec} \Rightarrow F$

$B \vdash \text{sigexp} \Rightarrow (T)E \quad B + T + \{\text{strid} \mapsto E\} \vdash \text{sigexp}' \Rightarrow (T')E'$

$F := \{\text{funid} \mapsto (T)(E, (T')E')\}$

$\langle B \vdash \text{funspec} \Rightarrow F', \text{funid} \notin \text{dom}(F')\rangle$

$B \vdash \text{funid(strid : sigexp)} : \text{sigexp}' \langle \text{and funspec} \rangle \Rightarrow F'(\langle + F' \rangle)$

$\Gamma \ ok$

\[
\begin{align*}
\text{tynames } F, G, E, U \subset T & \quad F, G, E, U \text{ wf } \\
(T, F, G, E), U \ ok & \quad \text{(49)}
\end{align*}
\]

$\Gamma \vdash \Upsilon : \text{Sig}$

\[
\begin{align*}
T &= T \text{ of } \Gamma \\
T \cap T' &= \emptyset & \text{tynames } F, G, E \subset T \cup T' \\
tyvars F, G, E &= \emptyset & F, G, E \text{ wf } \\
\Gamma \vdash (T')(F, G, E) : \text{Sig} & \quad \text{(50)}
\end{align*}
\]

$\Gamma \vdash \text{uspecs ok}$

$\Gamma \vdash \text{unitid} : \Upsilon, \text{uspecs ok}$

$\Upsilon = (T')(F, G, E) \quad \Gamma + T' \vdash \text{uspecs ok}$

$E : \Sigma \ using \ \varphi$

\[
\begin{align*}
\Sigma & \geq E^- \ using \ \varphi & E & \succ E^- \\
E & : \Sigma \ using \ \varphi & \quad \text{(53)}
\end{align*}
\]

$\Sigma \equiv \Sigma' \ using \ \varphi$

\[
\begin{align*}
T &= \varphi(T') & E & = \varphi(E') & \text{Supp } \varphi \subset T' & \#T = \#T' \\
(T)E & \equiv (T')E' \ using \ \varphi & \quad \text{(54)}
\end{align*}
\]

40
\[ \Upsilon \equiv \Upsilon' \text{ using } \varphi \]

\[ (T)E \equiv (T')E' \text{ using } \varphi \quad F = \varphi(F') \quad G = \varphi(G') \]

\[ (T)(F, G, E) \equiv (T')(F', G', E') \text{ using } \varphi \] (55)

\[ F, G, E : \Upsilon \text{ using } \varphi \]

\[ E : (T)E' \text{ using } \varphi \]
\[ \text{dom}(F) \supset \text{dom}(F') \]
\[ \forall \text{funid} \in \text{dom}(F'). \exists \varphi_1, \varphi_2 \]
\[ E_1' : (T_1)E_1 \text{ using } \varphi_1 \]
\[ \varphi_1(E_2) : (T_2')E_2' \text{ using } \varphi_2 \]

\[ \text{where } (T_1')E_1', (T_2')E_2' = \varphi(F'(\text{funid})) \]

\[ \text{and } (T_1)(E_1', (T_2'E_2') = F(\text{funid}) \] (56)

\[ \forall \text{sigid} \in \text{dom}(G'). \exists \varphi'. G(\text{sigid}) \equiv \varphi(G'(\text{sigid})) \text{ using } \varphi' \]

\[ F, G, E : (T')(F', G', E') \text{ using } \varphi \]
\[(F, G, E) \text{ or } B \in \text{ Basis } = \text{ FunEnv } \times \text{ SigEnv } \times \text{ Env}\]
\[F \in \text{ FunEnv } = \text{ FunId}^{\text{fin}} \rightarrow \text{ FunctorClosure}\]
\[G \in \text{ SigEnv } = \text{ SigId}^{\text{fin}} \rightarrow \text{ Int}\]
\[(SE, TE, VE) \text{ or } E \in \text{ Env } = \text{ StrEnv } \times \text{ TyEnv } \times \text{ ValEnv}\]
\[(\text{strid} : I, \text{strexp}, B) \in \text{ FunctorClosure } = (\text{StrId} \times \text{Int}) \times \text{StrExp} \times \text{Basis}\]
\[(SI, TI, VI) \text{ or } I \in \text{ Int } = \text{ StrInt } \times \text{TyInt } \times \text{ValInt}\]
\[SE \in \text{ StrEnv } = \text{ StrId}^{\text{fin}} \rightarrow \text{ Env}\]
\[TE \in \text{ TyEnv } = \text{ TyCon}^{\text{fin}} \rightarrow \text{ ValEnv}\]
\[VE \in \text{ ValEnv } = \text{ VId}^{\text{fin}} \rightarrow \text{ Val} \times \text{IdStatus}\]
\[SI \in \text{ StrInt } = \text{ StrId}^{\text{fin}} \rightarrow \text{ Int}\]
\[TI \in \text{ TyInt } = \text{ TyCon}^{\text{fin}} \rightarrow \text{ ValInt}\]
\[VI \in \text{ ValInt } = \text{ VId}^{\text{fin}} \rightarrow \text{ IdStatus}\]
\[(G, I) \text{ or } IB \in \text{ IntBasis } = \text{ SigEnv } \times \text{ Int}\]

Figure 21: Dynamic TDIL for Standard ML (summary)

D Unit Dynamic Semantic Rules

We use the following definitions and notation.

- We recall the dynamic TDIL for Standard ML in Figure 21.
- We extend the TDIL category \(\text{FunctorClosure}\) as follows, permitting an optional ascribed interface on the functor body.

\[
\text{FunctorClosure} = \text{FClos} \cup \text{FClos}'
\]
\[(\text{strid} : I, \text{strexp}, B) \in \text{FClos} = (\text{StrId} \times \text{Int}) \times \text{StrExp} \times \text{Basis}\]
\[(\text{strid} : I, \text{strexp} : I', B) \in \text{FClos}' = (\text{StrId} \times \text{Int}) \times (\text{StrExp} \times \text{Int}) \times \text{Basis}\].

Here, \(A \cup B\) denotes the disjoint union of \(A\) and \(B\).

- We define the function \(\downarrow: \text{Basis } \times \text{UnitInt } \rightarrow \text{Basis}\) that cuts down a basis \(B\) to match a unit interface \(UI\):

\[
\downarrow : \text{Basis } \times \text{UnitInt } \rightarrow \text{Basis}
\]
\[(F, G, E) \downarrow (FI, I) = (F \downarrow FI, \{\}, E \downarrow I)\]

\[
\downarrow : \text{FunEnv } \times \text{FunIntEnv } \rightarrow \text{FunEnv}
\]
\[F \downarrow FI = \{\text{funid} \mapsto F(\text{funid}) \downarrow FI(\text{funid}) : \text{funid} \in \text{dom}(F) \cap \text{dom}(FI)\}\]

\[
\downarrow : \text{FunctorClosure } \times (\text{Int } \times \text{Int}) \rightarrow \text{FunctorClosure}
\]
\[(\text{strid}:I_0, \text{strexp}, B) \downarrow (I, I') = (\text{strid}:I, \text{strexp}:I', B)\]
\[(\text{strid}:I_0, \text{strexp}:I'_0, B) \downarrow (I, I') = (\text{strid}:I, \text{strexp}:I'_0, B)\].

- TD’s evaluation judgement \(s, B \vdash \text{strexp} \Rightarrow E/p, s'\) handles functor application for functor closures of the form \(\text{FClos}\). We extend that judgement by adding the following rules to handle
functor closures of the form FClos′.

\[
\begin{align*}
B(\text{funid}) &= (\text{strid}:I, \text{strexpr}:I', B') \\
\Gamma, s, B \vdash \text{strexpr} \Rightarrow E, s' &\quad s', B', +\{\text{strid} \mapsto E \downarrow I\} \vdash \text{strexpr} \Rightarrow E', s'' \\
\Gamma, s, B \vdash \text{funid}(\text{strexpr}) \Rightarrow E' \downarrow I', s'' \\
B(\text{funid}) &= (\text{strid}:I, \text{strexpr}:I', B') \\
\Gamma, s, B \vdash \text{strexpr} \Rightarrow p, s' \\
\Gamma, s, B \vdash \text{funid}(\text{strexpr}) \Rightarrow p, s''
\end{align*}
\]

\[
\begin{align*}
\Gamma, s, \text{udecs} \Rightarrow \Gamma'/p, s' \\
\Gamma, s \vdash \cdot \Rightarrow \Gamma, s
\end{align*}
\]

(57)

\[
\begin{align*}
\Gamma, s \vdash \text{srcunit} = (\text{unit} \ \text{unitid} = \text{unit} \ \text{topdec} \ \text{end}) \\
\Gamma, s, \text{udecs} \Rightarrow \Gamma', s''
\end{align*}
\]

(58)

\[
\begin{align*}
\Gamma, s \vdash \text{srcunit} = (\text{unit} \ \text{unitid} = \text{unit} \ \text{topdec} \ \text{end}) \\
\Gamma, s \vdash \text{srcunit}, \text{udecs} \Rightarrow \Gamma', s''
\end{align*}
\]

(59)

\[
\begin{align*}
\Gamma, s \vdash \text{srcunit} = (\text{unit} \ \text{unitid} = \text{unit} \ \text{topdec} \ \text{end}) \\
\Gamma, s \vdash \text{srcunit}, \text{udecs} \Rightarrow p, s'
\end{align*}
\]

(60)

\[
\begin{align*}
\Gamma \vdash \text{impexp} \Rightarrow B \\
\Gamma, s \vdash \text{import impexp} \Rightarrow B, s
\end{align*}
\]

(61)

\[
\begin{align*}
\Gamma, s \vdash \text{strdec} \Rightarrow E, s' \\
\Gamma, s \vdash \text{strdec} \Rightarrow \{\}, \{\}, E, s'
\end{align*}
\]

(62)

\[
\begin{align*}
\Gamma, s \vdash \text{strdec} \Rightarrow p, s' \\
\Gamma, s \vdash \text{strdec} \Rightarrow p, s'
\end{align*}
\]

(63)

\[
\begin{align*}
\Gamma \vdash \text{sigdec} \Rightarrow G \\
\Gamma, s \vdash \text{sigdec} \Rightarrow \{\}, G, E_0, s
\end{align*}
\]

(64)
\[ B \vdash \text{fundec} \Rightarrow F \]
\[ s, (B, U) \vdash \text{fundec} \Rightarrow (F, \{\}, E_{\emptyset}), s \] (65)

\[ s, \Gamma \vdash \text{topdec} \Rightarrow B, s', \Gamma + B \vdash \text{topdec}' \Rightarrow B', s'' \]
\[ s, \Gamma \vdash \text{local topdec in topdec'} \text{ end} \Rightarrow B', s'' \] (66)

\[ s, \Gamma \vdash \text{topdec} \Rightarrow p, s' \]
\[ s, \Gamma \vdash \text{local topdec in topdec'} \text{ end} \Rightarrow p, s' \] (67)

\[ s, \Gamma \vdash \text{topdec} \Rightarrow B, s', \Gamma + B \vdash \text{topdec}' \Rightarrow p, s'' \]
\[ s, \Gamma \vdash \text{local topdec in topdec'} \text{ end} \Rightarrow p, s'' \] (68)

\[ s, \Gamma \vdash \text{topdec} \Rightarrow B, s', \Gamma + B \vdash \text{topdec}' \Rightarrow B', s'' \]
\[ s, \Gamma \vdash \text{topdec topdec'} \Rightarrow B + B', s'' \] (69)

\[ s, \Gamma \vdash \text{topdec} \Rightarrow p, s' \]
\[ s, \Gamma \vdash \text{topdec topdec'} \Rightarrow p, s' \] (70)

\[ s, \Gamma \vdash \text{topdec} \Rightarrow B, s', \Gamma + B \vdash \text{topdec}' \Rightarrow p, s'' \]
\[ s, \Gamma \vdash \text{topdec topdec'} \Rightarrow p, s'' \] (71)

\[ \Gamma \vdash \text{impexp} \Rightarrow B \]

\[ \Gamma(\text{unitid}) = B' \]
\[ \Gamma \vdash \text{unitid} \Rightarrow B' \] (72)

\[ \text{Inter} B \vdash \text{topspec} \Rightarrow \text{UI} \quad U(\text{unitid}) = B' \]
\[ B, U \vdash \text{unitid} : \text{intf topspec} \text{ end} \Rightarrow B' \downarrow \text{UI} \] (73)

\[ \Gamma \vdash \text{impexp} \Rightarrow B \quad \Gamma \vdash \text{impexp}' \Rightarrow B' \]
\[ \Gamma \vdash \text{impexp impexp'} \Rightarrow B + B' \] (74)

\[ IB \vdash \text{topspec} \Rightarrow \text{UI} \]

\[ IB \vdash \text{spec} \Rightarrow I \]
\[ IB \vdash \text{spec} \Rightarrow \{\}, I \] (75)

\[ IB \vdash \text{funspec} \Rightarrow FI \]
\[ IB \vdash \text{functor funspec} \Rightarrow FI, \{\}, \{\}, \{\} \] (76)

\[ IB \vdash \text{topspec} \Rightarrow FI, I \quad IB + I \vdash \text{topspec'} \Rightarrow FI', I' \]
\[ IB \vdash \text{topspec topspec'} \Rightarrow FI + FI', I + I' \] (77)
E TD Linking Rules

TD defines the initial static basis $B_0 = T_0, F_0, G_0, E_0$ [16, Appendix C]. We define the basis interface

$$
\Upsilon_{basis} = (T_0)(F_0, G_0, E_0)
$$

and assume the unit identifier basis may not appear in source units.

$$
\Gamma \vdash L \ ok
$$

$$
\Gamma \vdash uspecs \ ok
\Gamma + uspecs \vdash exps \ ok
uspecs = unitid_1: \Upsilon_1, \ldots, unitid_n: \Upsilon_n
unitid_1, \ldots, unitid_n \ are \ distinct
\Gamma \vdash uspecs \rightarrow exps \ ok
$$

$$
\Gamma \vdash exps \ ok
$$

$$
\Gamma \vdash udecs : uspecs
\Gamma \vdash udecs : uspecs \ ok
$$

$$
\Gamma \vdash (T)(F, G, E) : \text{Sig}
\Gamma \vdash T, F, G, E \ ok
$$

$$
\Gamma \vdash udecs
$$

$$
\cdot \rightarrow udecs : uspecs \Rightarrow udecs
$$

$$
\Upsilon = \Upsilon_{basis} \ using \ \varphi
basis: \Upsilon \rightarrow udecs : uspecs \Rightarrow udecs
$$

$$
\Gamma \vdash L \Rightarrow udecs
$$

$$
\Gamma \vdash L\Rightarrow L' \Rightarrow L''
$$

$$
\Gamma \vdash exps \Rightarrow exps'
\Gamma \vdash L\Rightarrow (uspecs_0 \rightarrow exps)\Rightarrow (\cdot \rightarrow exps') \Rightarrow uspecs_0 \rightarrow exps''
$$
unitid ∈ dom(uspecs)
(Γ + uspecs_0 + uspecs)(unitid) = T', F', G', E'
F', G', E' : Υ using ϕ
L' := ϕ(uspecs_1 → udecs' : uspecs')
Γ ⊢ (uspecs_0 → udecs : uspecs)++L' ⇒ L''

Γ ⊢ (uspecs_0 → udecs : uspecs)++(unitid:Υ, uspecs_1 → udecs' : uspecs') ⇒ L''

unitid ∉ dom(exps)
uspecs_0 = uspecs'', unitid:Υ'', uspecs'''
Υ'' ≡ Υ using ϕ
L' := ϕ(uspecs_1 → exps')
Γ ⊢ (uspecs_0 → exps)++L' ⇒ L''

Γ ⊢ (uspecs_0 → exps)++(unitid:Υ, uspecs_1 → exps') ⇒ L''

Γ ⊢ exps++exps' ⇒ exps''

Γ ⊢ (udecs : uspecs)++(udecs' : uspecs') ⇒ udecs, udecs' : uspecs, uspecs'

Γ ⊢ B++B' ⇒ B + B'

F TD Elaboration Rules
We define the bound type names in a linkset, BT(L), by

BT(uspecs_0 → udecs : uspecs) = BT(uspecs_0) ∪ BT(uspecs)
BT(uspecs_0 → B) = BT(uspecs_0).

Γ ⊢ L
\[ \text{project} \Rightarrow L \quad \Upsilon_{\text{basis}} = (T)(F, G, E) \quad T \cap \text{BT}(L) = \emptyset \]

\[ L = \text{uspecs}_0 \rightarrow \text{udecs} : \text{uspecs} \]

\[ \Gamma := ((T, F, G, E), \{\}) + \text{BT}(\text{uspecs}_0) + \text{uspecs} \]

\[ \Gamma \vdash \text{srcunit} \Rightarrow \text{uspecs}_1 \rightarrow \text{udecs}' : \text{uspecs}' \]

\[ \vdash L \vdash (\text{basis}: \Upsilon_{\text{basis}}, \text{uspecs}_1 \rightarrow \text{udecs}' : \text{uspecs}') \Rightarrow L'' \]

\[ \text{project, srcunit} \Rightarrow L'' \]

\[ \text{project} \Rightarrow L \]

\[ \text{BT}(L) \cap \text{BT}(L') = \emptyset \quad \vdash L' \text{ ok} \]

\[ \vdash L \vdash L' \Rightarrow L'' \]

\[ \text{project, L'} \Rightarrow L'' \]

\[ \Gamma \vdash \text{srcunit} \Rightarrow L \]

\[ \text{srcunit} = (\text{unit unitid} = \text{unit topdec end}) \]

\[ \Gamma \vdash \text{topdec} \Rightarrow \text{uspecs}_0 \rightarrow T, F, G, E \]

\[ \Gamma \vdash \text{srcunit} \Rightarrow \text{uspecs}_0 \rightarrow \text{srcunit} : \text{unitid}: (T)(F, G, E) \]

\[ \Gamma \vdash \text{topdec} \Rightarrow L \]

\[ \Gamma \vdash \text{impexp} \Rightarrow L \]

\[ \Gamma \vdash \text{import impexp} \Rightarrow L \]

\[ \text{topdec} = \text{strdec} \quad \Gamma \vdash \text{topdec} : B \]

\[ \Gamma \vdash \text{topdec} \Rightarrow \cdot \rightarrow B \]

\[ \text{topdec} = \text{sigdec} \quad \Gamma \vdash \text{topdec} : B \]

\[ \Gamma \vdash \text{topdec} \Rightarrow \cdot \rightarrow B \]

\[ \text{topdec} = \text{fundec} \quad \Gamma \vdash \text{fundec} : B \]

\[ \Gamma \vdash \text{topdec} \Rightarrow \cdot \rightarrow B \]

\[ \Gamma \vdash \text{topdec} \Rightarrow \text{uspecs}_0 \rightarrow B \]

\[ \Gamma + \text{BT}(\text{uspecs}_0) + B \vdash \text{topdec}' \Rightarrow L \]

\[ B' := \text{T of } B, \{\}, \{\}, E_0 \quad \Gamma \vdash (\text{uspecs}_0 \rightarrow B') \vdash L \Rightarrow L' \]

\[ \Gamma \vdash \text{local topdec in topdec' end} \Rightarrow L' \]

\[ \Gamma \vdash \text{topdec} \Rightarrow L \]

\[ L = \text{uspecs}_0 \rightarrow B \]

\[ \Gamma + \text{BT}(\text{uspecs}_0) + B \vdash \text{topdec}' \Rightarrow L' \]

\[ \Gamma \vdash L \vdash L' \Rightarrow L'' \]

\[ \Gamma \vdash \text{topdec topdec'} \Rightarrow L'' \]
$$\Gamma \vdash \text{impexp} \Rightarrow L$$

$$\Gamma(\text{unitid}) = T, F, G, E \quad L := \text{unitid}:(T)(F, G, E) \rightarrow \emptyset, F, G, E$$

$$\Gamma \vdash \text{unitid} \Rightarrow L$$

$$\Gamma \vdash \text{intexp} \Rightarrow (T)(F, G, E) \quad L := \text{unitid}:(T)(F, G, E) \rightarrow \emptyset, F, G, E$$

$$\Gamma \vdash \text{unitid} : \text{intexp} \Rightarrow L$$

$$\Gamma \vdash \text{impexp} \Rightarrow L \quad \Gamma \vdash \text{impexp}' \Rightarrow L'$$

$$\text{BT}(L) \cap \text{BT}(L') = \emptyset \quad \Gamma \vdash L + L' \Rightarrow L''$$

$$\Gamma \vdash \text{impexp \ impexp}' \Rightarrow L''$$
In this appendix we present a more realistic example based on code used in Frank Pfenning’s 1995 Logic Programming course at Carnegie Mellon. The example comprises a stream library, an input library, a parsing library, and an interpreter for Mini-ML that uses the libraries.\(^9\) In Figure 22, we illustrate the high-level dependencies in the code. We first discuss the library handoff units and the interpreter. Then we give library implementations and show how to link these to make an executable interpreter.

Stream Library

The stream library declares a signature \texttt{BASIC\_STREAM} describing the visible “core” of functional streams, a signature \texttt{STREAM} that extends \texttt{BASIC\_STREAM}, a functor \texttt{Stream} that lifts a \texttt{BASIC\_STREAM} structure to \texttt{STREAM}, and plain and memoizing \texttt{STREAM} structures. The handoff unit \texttt{StreamLib} declares all of these components by importing units \texttt{STREAMSIG} and \texttt{StreamImpl}:

\begin{verbatim}
unit STREAMSIG =
  unit
  signature BASIC\_STREAM =
    sig
      type 'a stream
      val empty : 'a stream
      val create : (unit -> ('a * 'a stream) option) -> 'a stream
      val expose : 'a stream -> ('a * 'a stream) option
      (* · · · *)
    end
\end{verbatim}

\(^9\)The libraries were written by Chris Okasaki, Frank Pfenning, and Bob Harper. Pfenning wrote the interpreter to demonstrate the libraries to teams of students using them to implement compilers for the course.
signature STREAM =
  sig
    include BASIC_STREAM
    exception Empty
    val delay : (unit -> 'a stream) -> 'a stream
    val lcons : 'a * (unit -> 'a stream) -> 'a stream
    val cons : 'a * 'a stream -> 'a stream
    val map : ('a -> 'b) -> 'a stream -> 'b stream
    (* ... *)
  end
end

unit StreamLib =
unit
  import STREAMSIG
import StreamImpl:
int
  functor Stream(structure BasicStream : BASIC_STREAM) :
    STREAM where type 'a stream = 'a BasicStream.stream
  structure PlainStream : STREAM
  structure MemoStream : STREAM
    (* default stream package memoizes *)
  structure Stream : STREAM
    where type 'a stream = 'a MemoStream.stream
end

The other libraries and the interpreter import StreamLib, heavily using its structure Stream : STREAM but never directly referring to the auxiliary units STREAMSIG and StreamImpl. We shall give a stream implementation StreamImpl that, like StreamLib, performs IC against STREAMSIG to obtain the stream signatures.

**Input Library**

The input library declares a signature INPUT and structure Input : INPUT. Structure Input creates character streams. For example, the function

    Input.promptkeybd : string -> char Stream.stream

creates a stream that, when forced, prompts the user for input a line at a time. The handoff unit InputLib imports units INPUTSIG and InputImpl:
This description of the input library performs IC against the stream library description (unit StreamLib) but SC against its implementation (unit StreamImpl).

Parsing Library

The parsing library declares a signature POS and structure Pos : POS for positions within a file, a signature BASIC_PARSING describing a “core” set of parsing combinators, a signature PARSING that extends BASIC_PARSING with additional combinators and utilities, a functor Parsing that lifts a BASIC_PARSING structure to PARSING, and, finally, a structure Parsing : PARSING. The parsing library uses the stream library. For example,

\[
\text{Pos.markstream : char Stream.stream } \rightarrow \text{ (char * Pos.pos) Stream.stream}
\]

marks a character stream with position information for reporting errors and

\[
\text{Parsing.transform : ('}a, 't) \text{ Parsing.parser } \rightarrow \text{ ('}t * \text{ Pos.pos) Stream.stream } \rightarrow \text{'a Stream.stream}
\]

creates a stream of parsed values from a parser and a marked stream of tokens. Following the now familiar pattern, the handoff unit ParsingLib imports PARSINGSIG and ParsingImpl:
unit PARSINGSIG =
  unit
  local
    import StreamLib
  in
    signature POS =
      sig
        type pos
        val markstream : char Stream.stream -> (char * pos) Stream.stream (* ... *)
        val makestring : pos -> string
      end
    signature BASIC_PARSING =
      sig
        (* Parser with token type 't, result type 'a *)
        type ('a,'t) parser
        type pos (* = Pos.pos *)
        val succeed : 'a -> ('a,'t) parser
        val fail : ('a,'t) parser
        val done : 'a -> ('a,'t) parser
        val any : ('t,'t) parser
        val -- : ('a,'t) parser * ('a -> ('b,'t) parser) -> ('b,'t) parser
        val ## : ('a,'t) parser * (pos -> ('a,'t) parser) -> ('a,'t) parser
        val !! : ('a,'t) parser -> ('a * pos,'t) parser
        val $ : (unit -> ('a,'t) parser) -> ('a,'t) parser
        val lookahead : ('a,'t) parser -> ('a -> ('b,'t) parser) -> ('b,'t) parser
        val parse : ('a,'t) parser -> ('t * pos) Stream.stream -> 'a option
        val transform : ('a,'t) parser -> ('t * pos) Stream.stream -> 'a Stream.stream
      end
    signature PARSING =
      sig
        include BASIC_PARSING
        val && : ('a,'t) parser * ('b,'t) parser -> ('a * 'b,'t) parser
        val || : ('a,'t) parser * ('a,'t) parser -> ('a,'t) parser
          (* ... many more ... *)
      end
  end
end

52
unit ParsingLib =
  unit
  import PARSINGSIG
  import ParsingImpl :
    intf
      structure Pos : POS
      functor Parsing(structure BasicParsing : BASIC_PARSING
        where type pos = Pos.pos)
        : PARSING
        where type 'a parser = 'a BasicParsing.parser
        where type pos = Pos.pos
      structure Parsing : PARSING
        where type pos = Pos.pos
      end
    end

Interpeter
The Mini-ML interpreter is a single unit Interp that declares signatures and structures for identifiers, lexical tokens, an abstract syntax, lexical analysis, parsing, evaluation, and a read-eval-print loop. It performs IC against the handoff units StreamLib, InputLib, and ParsingLib and units BoolLib and IntLib (not shown but assumed to declare the Standard Basis Library structures Bool and Int).

unit Interp =
  unit
  import StreamLib InputLib ParsingLib IntLib BoolLib
  signature ID =
    sig
      eqtype id
      val id : string -> id
      val eq : id * id -> bool
      val makestring : id -> string
    end

    structure Id := ID = struct (* · · · *) end
signature TOKEN =
  sig
  datatype token =
    Id of Id.id
  | Num of int
  | LParen | RParen
  | Plus | Times | Neg | Eql
  | Fn | Rec | Is
  | LAngle | RAngle | Comma | Fst | Snd
  | True | False | If | Then | Else
  | Let | Be | In
  | Semi

  val makestring : token -> string
end

structure Token :> TOKEN = struct (* · · · *) end

signature ABSSYN =
  sig
  datatype abssyn =
    Var of Id.id
  | Num of int
  | Bool of bool
  (* · · · *)

  val makestring : abssyn -> string
end

structure AbsSyn :> ABSSYN = struct (* · · · *) end

signature LEXER =
  sig
  val lex_item : (Token.token * Pos.pos, char) Parsing.parser
  val lex :
    (char * Pos.pos) Stream.stream ->
    (Token.token * Pos.pos) Stream.stream
  end

structure Lexer :> LEXER = struct (* · · · *) end

signature PARSER =
  sig
    val parse_prog : (AbsSyn.abssyn, Token.token) Parsing.parser
    val parse :
      (Token.token * Pos.pos) Stream.stream
      -> AbsSyn.abssyn Stream.stream
  end

structure Parser :> PARSER = struct (* · · · *) end
signature EVALUATOR =
  sig
    val evaluate : AbsSyn.abssyn Stream.stream
    -> AbsSyn.abssyn Stream.stream
  end

structure Evaluator :> EVALUATOR = struct (* · · · *) end

signature INTERPRETER =
  sig
    val interpreter : char Stream.stream -> string Stream.stream
    val interact : unit -> unit
  end

structure Interpreter :> INTERPRETER =
  struct
    val interpreter =
      (Stream.map AbsSyn.makestring) o
      (Evaluator.evaluate) o
      (Parser.parse) o
      (Lexer.lex) o
      (Pos.markstream)
    fun interact () =
      let
        fun loop s =
          (case Stream.expose s of
            NONE => ()
            | SOME (result, s') =>
              (print (result ^ "\n"); loop s'))
        val is = interpreter (Input.promptkeybd ">")
      in
        loop is
      end

end

The interpreter is organized around the stream library. For example, the Mini-ML evaluator has type

Evaluator.evaluate : AbsSyn.abssyn Stream.stream
  -> AbsSyn.abssyn Stream.stream

and maps a stream of expressions into a stream of values, filtering out those expressions whose evaluation gets stuck (and printing an error message). This organization lends itself well to experimenting and debugging. A one-line change
val interpreter = 
    (Stream.map AbsSyn.makestring) o 
    (* (Evaluator.evaluate) o *) 
    (Parser.parse) o 
    (Lexer.lex) o 
    (Pos.markstream)

produces a variant of the interpreter that accepts syntactically correct Mini-ML expressions and 
prints the resulting abstract syntax.

Library Implementations

Unit StreamImpl performs IC against the unit STREAMSIG:

```plaintext
unit StreamImpl =
    unit
    import STREAMSIG

functor Stream(structure BasicStream : BASIC_STREAM) :
    > STREAM where type 'a stream = 'a BasicStream.stream =
    struct
    open BasicStream
    exception Empty
    fun delay t = create (expose o t)
    (* ... *)
    end

structure BasicStream :> BASIC_STREAM =
    struct (* ... *) end

structure PlainStream : STREAM =
    Stream (structure BasicStream = BasicStream)

structure BasicMemoStream :> BASIC_STREAM =
    struct (* ... *) end

    (* default stream package memoizes *)
    structure Stream = MemoStream
end
```

Unit InputImpl performs IC against INPUTSIG and StreamLib but SC against StreamImpl. It 
also needs a way to perform IO: We assume the unit TextIOLib (not shown) declares the Standard 
Basis Library structure TextIO.

```plaintext
unit InputImpl =
    unit
    local
    import INPUTSIG StreamLib TextIOLib
    in
    structure Input :> INPUT = struct (* ... *) end
    end
end
```
We declared everything specified by StreamLib in StreamImpl. Here we demonstrate an alternative. Unit ParsingImpl imports structure Pos and functor Parsing from auxiliary units PosImpl and ParsingFunImpl:

unit PosImpl =
unit
  local
    import PARSINGSIG StreamLib
    in
      structure Pos :> POS = struct (* · · · *) end
  end
end

unit ParsingFunImpl =
unit
  local
    import PARSINGSIG StreamLib PosImpl
    in
      functor Parsing (structure BasicParsing : BASIC_PARSING
        where type pos = Pos.pos) : PARSING =
        struct (* · · · *) end
    end
  end
end

unit ParsingImpl =
unit
  local
    import PARSINGSIG StreamLib
    in
      import PosImpl ParsingFunImpl
      structure BasicParsing :> BASIC_PARSING where type pos = Pos.pos =
        struct (* · · · *) end
      structure Parsing =
        Parsing (structure BasicParsing = BasicParsing)
  end
end

Linking
To create an executable interpreter, we shall link unit Interp with library implementations and the unit

unit RunInterp =
unit
  import Interp
  val () = Interpreter.interact()
end
that gets things started. We begin by compiling \texttt{Interp} and \texttt{RunInterp}:

\begin{verbatim}
L_r = link(STREAMSIG, StreamLib,
            INPUTSIG, InputLib,
            PARSINGSIG, ParsingLib,
            IntLib, BoolLib,
            Interp,
            RunInterp).
\end{verbatim}

We then compile the libraries:

\begin{verbatim}
LS = link(STREAMSIG, StreamImpl)
LI = link(STREAMSIG, StreamLib, INPUTSIG, TextIOLib, InputImpl)
LP = link(STREAMSIG, StreamLib,
          PARSINGSIG, PosImpl, ParsingFunImpl, ParsingImpl)
\end{verbatim}

Finally, we complete the linkset

\begin{verbatim}
L = link(LS, LI, LP, L_r)
\end{verbatim}

to an executable program. When run, the program will prompt the user, parse Mini-ML expressions from its input, evaluate these in turn, and print the resulting values (provided evaluation terminates). The following typescript demonstrates the program in action.

\begin{verbatim}
> true+3;
  Run-time error arithmetic type error
> let sq be fn x in x*x in sq(sq 4);
    256
> let fib be rec f(x) is
>    if x=0 then 1
>    else if x=1 then 1
>    else f(x + ~1) + f(x + ~2)
  > in fib 7;
    21
> 
\end{verbatim}

An implementation could avoid unnecessary recompilation, maintaining a repository of compiled code during IC. For example, such an implementation might employ

\begin{verbatim}
L_0 = link(STREAMSIG)
L_1 = link(L_0, StreamImpl)
L_2 = link(L_0, StreamLib)
L_3 = link(L_2, INPUTSIG, TextIOLib, InputImpl)
L_4 = link(L_2, PARSINGSIG, PosImpl, ParsingFunImpl, ParsingImpl)
\end{verbatim}

to avoid recompiling STREAMSIG and StreamLib while producing \( L_S = L_1, L_I = L_3, \) and \( L_P = L_4. \)