A Separate Compilation Extension to Standard ML

David Swasey Tom Murphy VII Karl Crary Robert Harper

Carnegie Mellon

{swasey,tom7,crary,rwh}@cs.cmu.edu

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.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax

General Terms programming languages, modularity, compilation

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. The semantics, formulated in the framework of the Harper-Stone semantics of ML [12, 13], is given in Section 3.[1] Some implementation issues are discussed in Section 4. We conclude with a discussion of related work in Section 6.

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.[2] However, since Standard ML lacks syntactically expressible 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.

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

[1] We plan to give a semantics in the framework of *The Definition of Standard ML* [16] in a forthcoming technical report [21].

[2] Consequently, units cannot be identified with Standard ML structures.

$srcunit$	$::=$	`unit` $unitid$ = `top` $topdec$ `end`	unit declaration
$topdec$	$::=$	`import` $impexp$	open units
		$strdec$	structure-level declaration
		$sigdec$	signature declaration
		$fundec$	functor declaration
		`local` $topdec_1$ `in` $topdec_2$ `end`	local declaration
		$topdec_1\ topdec_2$	
$impexp$	$::=$	$unitid\ \langle :\ intexp \rangle$	open unit $unitid$
		$impexp_1\ impexp_2$	
$intexp$	$::=$	`intf` $topspec$ `end`	interface expression
$topspec$	$::=$	$spec$	structure-level specification
		`functor` $funspec$	functor specification
		$topspec_1\ topspec_2$	
$funspec$	$::=$	$funid(strid : sigexp) : sigexp'$ $\langle$`and` $funspec\rangle$	

Figure 1. SMLSC concrete syntax

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.

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 `import`s, just as the structure declaration of the same name.

Projects and Linksets

A *project* consists of a linearly ordered sequence of source units and compiled units, which we call *linksets* (following Cardelli [5]). 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.

Each linkset consists of several compiled units, called its exports, together with the names and interfaces of its imports, the units on which it depends. 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 Section 3.

Examples

We begin with a few simple examples to illustrate the features of the system, building up to the idiom of *handoff units*.

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 =
top
  signature QUEUE =
    sig
       type α queue
       val empty : α queue
       val push : α * α queue -> α
       val pop : α queue -> α * α queue
    end
  structure Queue :> QUEUE =
    struct (* ... *) end
end
```

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

```
unit Scheduler =
top
  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

$$\begin{aligned} L_0 &= link(Collections) \\ L_1 &= link(L_0, Scheduler) \end{aligned}$$

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 =
top
  import Collections :
    intf
      structure Queue :
        sig
          type α queue
          val empty : α queue
          val push : α * α queue -> α
          val pop : α queue -> α * α 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:

$$\begin{aligned} L_0 &= link(Scheduler2) \\ L_1 &= link(Collections) \\ L_2 &= link(L_1, L_0) \end{aligned}$$

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 =
top
  signature QUEUE =
    sig
       type α queue
       val empty : α queue
       val push : α * α queue -> α
       val pop : α queue -> α * α queue
    end
  import CollectionsImpl :
    intf
      structure Queue : QUEUE
    end
end
```

The implementation of the collections library is moved to the unit `CollectionsImpl`. Because the `import` declaration opens the imported unit, all of the contents of `CollectionsImpl` are available in the `Collections` unit. (Note that signature declarations do not appear in interfaces, but we can write these before the import declaration and then use the signature bindings in the interface for the same effect.) Clients wishing to make use of the library simply import the handoff unit using IC:

```
unit Scheduler3 =
top
  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:

$$L_0 = link(Collections, Scheduler3, OtherClient)$$

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

$$L_1 = link(CollectionsImpl, L_0)$$

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 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. No additional sharing specifications are required to use the separate compilation system.

This is in sharp contrast to the so-called *fully functorized style* of use of the ML modules system, in which a functor is λ-abstracted over the modules on which it depends. In this formulation references are interpreted as *indefinite*, in that the functors involved may be applied to *any* modules satisfying those interfaces, not necessarily in a coherent manner. Different functors with a parameter referring to "the" shared library might be applied to different instances of it. To ensure coherence in the presence of indefinite references, one must resort to explicit type sharing specifications, which are potentially burdensome.

The handoff methodology in SMLSC facilitates programming with definite references. Two pieces of code that SC import the same unit 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 5.

3. Semantics

We give a semantics to SMLSC by extending Harper and Stone's Typed Semantics (TS) for Standard ML [13].[3] 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 knd. 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 module, 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 $decs = dec_1, \ldots, dec_n$ declares expression ($var{:}con$), constructor ($var{:}knd\langle = con\rangle$), and module ($var{:}sig$) variables. A structure declaration list $sdecs$ has the form

$$lab_1 \triangleright dec_1, \ldots, lab_n \triangleright dec_n$$

associating a label with each declaration. The structure declaration list $lab \triangleright dec, sdecs$ 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:

$$m : [T \triangleright t{:}\Omega, X \triangleright x{:}t].$$

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.

[3] In a planned technical report [21], we intend to also give a semantics by extending *The Definition of Standard ML* [16].

Judgement...	*Meaning...*
$\vdash decs$ ok	$decs$ is well-formed
$decs \vdash sdecs$ ok	$sdecs$ is well-formed
$decs \vdash sig : \mathsf{Sig}$	sig is well-formed
$decs \vdash sig \equiv sig' : \mathsf{Sig}$	signature equivalence
$decs \vdash sbnds : sdecs$	$sbnds$ has declaration list $sdecs$
$decs \vdash mod : sig$	mod has signature sig

Figure 2. TSIL judgements (summary)

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

$$lab_1 \triangleright bnd_1, \ldots, lab_n \triangleright bnd_n.$$

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.

- $decs \vdash sdecs$ ok. No label is used twice and every declaration is well-formed. For example,

$$\vdash T \triangleright t{:}\Omega, X \triangleright x{:}t \text{ ok}.$$

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

$$decs \vdash (lab \triangleright var{=}mod, sbnds) : (lab \triangleright var{:}sig, sdecs)$$

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

$$m : [T \triangleright t{:}\Omega, X \triangleright x{:}t] \vdash m : [T \triangleright t{:}\Omega{=}m.T, X \triangleright x{:}t]$$

and

$$m : [T \triangleright t{:}\Omega, X \triangleright x{:}t] \vdash m : [T \triangleright t{:}\Omega, X \triangleright x{:}t].$$

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 Γ 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:

- $\Gamma \vdash strdec \rightsquigarrow sbnds : sdecs$. Elaborate the TSEL structure declaration $strdec$ to the structure binding list $sbnds : sdecs$. Since the TSEL permits functors within structures, this includes elaboration of functor declarations.
- $\Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig}$. Elaborate the TSEL signature expression $sigexp$ to the signature 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.

Judgement…	*Meaning…*
$\Gamma \vdash \mathit{strdec} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}$	structure declaration elaboration
$\Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig}$	signature elaboration
$\Gamma \vdash \mathit{spec} \rightsquigarrow \mathit{sdecs}$	specification elaboration
$\Gamma \vdash_{\mathsf{ctx}} \mathit{labs} \rightsquigarrow \mathit{path} : \mathit{class}$	context lookup
$\mathit{decs} \vdash_{\mathsf{sub}} \mathit{path} : \mathit{sig}_0 \preceq \mathit{sig} \rightsquigarrow \mathit{mod} : \mathit{sig}'$	coercion compilation

Figure 3. TS elaboration judgements (summary)

- $\Gamma \vdash \mathit{spec} \rightsquigarrow \mathit{sdecs}$. Elaborate the TSEL specification spec to the structure declaration list sdecs. This includes elaboration of functor specifications.
- $\Gamma \vdash_{\mathsf{ctx}} \mathit{labs} \rightsquigarrow \mathit{path} : \mathit{class}$. Perform identifier resolution in the context Γ. 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 class. Some labels in the context are annotated with a star, indicating that they are "open" (in the sense of the SML `open` declaration). Identifier resolution searches Γ from right to left, descending into structures with starred labels. For example, we may derive
$$T \triangleright t_1{:}\Omega, T \triangleright t_2{:}\Omega{=}\{\} \vdash T \rightsquigarrow t_2 : \Omega{=}\{\}$$
and
$$X \triangleright x_1{:}\{\}, 1^\star \triangleright m{:}[T \triangleright t{:}\Omega, X \triangleright x_2{:}t] \vdash X \rightsquigarrow m.X : m.T.$$
- $\mathit{decs} \vdash_{\mathsf{sub}} \mathit{path} : \mathit{sig}_0 \preceq \mathit{sig} \rightsquigarrow \mathit{mod} : \mathit{sig}'$. Perform transparent signature ascription. The inputs are a signature sig_0, a path having that signature, and a target signature sig. The output is a module $\mathit{mod} : \mathit{sig}'$, where sig' has the same shape as sig but is fully transparent relative to path.

Elaboration maps TSEL identifiers to TSIL labels using a function $\overline{\cdot}$. To implement identifier "shadowing," elaboration employs a function $\mathit{sbnds}{+\!\!+}\mathit{sbnds}' : \mathit{sdecs}{+\!\!+}\mathit{sdecs}'$ that concatenates $\mathit{sbnds} : \mathit{sdecs}$ and $\mathit{sbnds}' : \mathit{sdecs}'$, 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
$$\begin{array}{lcl} \mathit{sbnds} : \mathit{sdecs} & = & \overline{T} \triangleright t_1{=}\{\} : \overline{T} \triangleright t_1{:}\Omega{=}\{\} \\ \mathit{sbnds}' : \mathit{sdecs}' & = & \overline{T} \triangleright t_2{=}\mathtt{Int} : \overline{T} \triangleright t_2{:}\Omega{=}\mathtt{Int}, \end{array}$$
then $\mathit{sbnds}{+\!\!+}\mathit{sbnds}' : \mathit{sdecs}{+\!\!+}\mathit{sdecs}'$ might be
$$(\mathit{lab} \triangleright t_1{=}\{\}, \overline{T} \triangleright t_2{=}\mathtt{Int}) : (\mathit{lab} \triangleright t_1{:}\Omega{=}\{\}, \overline{T} \triangleright t_2{:}\Omega{=}\mathtt{Int})$$
where lab is not in the range of the $\overline{\cdot}$ function.

3.1 Linking

We define linking for the TSIL by giving rules for deriving the linking judgements in Figure 4. A linkset
$$\mathit{sdecs}_0 \rightarrow \mathit{sbnds} : \mathit{sdecs}; S$$
comprises imports sdecs_0, exports $\mathit{sbnds} : \mathit{sdecs}$, and signature abbreviations S.

- 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
$$\begin{array}{lcl} \mathit{sdecs}_{AB} & = & A \triangleright a{:}[T \triangleright t{:}\Omega, X \triangleright x{:}t], \\ & & B \triangleright b{:}[Y \triangleright y{:}a.T] \end{array}$$
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 $\mathit{sbnds} : \mathit{sdecs}$ are the TSIL code associated with the linkset. They may make reference to the linkset's imports. Continuing our example, the exports
$$\begin{array}{lcl} \mathit{sbnds}_{ZR} : \mathit{sdecs}_{ZR} & = & Z \triangleright z{=}b.Y, R \triangleright r{=}a.T : \\ & & Z \triangleright z{:}a.T, R \triangleright r{:}\Omega{=}a.T \end{array}$$
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
$$S_{\mathtt{SIG}} \;=\; \mathtt{SIG}{=}[M \triangleright m{:}\Omega{=}r]$$
specify that elaboration should treat the signature identifier `SIG` as an abbreviation for a TSIL signature referencing the exported type R.

Judgement…	*Meaning…*
$\mathit{decs} \vdash L\ \mathsf{ok}$	L is well-formed
$\mathit{decs} \vdash S\ \mathsf{ok}$	S is well-formed
$L \rightsquigarrow \mathit{exp} : \{\}$	L completes to exp
$\mathit{decs} \vdash L{+\!\!+}L' \rightsquigarrow L''$	L and L' merge to L''

Figure 4. Linking judgements

L	$::=$	$\mathit{sdecs}_0 \rightarrow \mathit{sbnds} : \mathit{sdecs}; S$	linkset
S	$::=$	$\cdot$	
		$S, \mathit{sigid} = \mathit{sig}$	top-level
		$S, \mathit{unitid} = S'$	declared by unitid

Figure 5. Linkset syntax

The dynamic semantics for SMLSC is very simple. The completion judgment $L \rightsquigarrow \mathit{exp} : \{\}$ translates a linkset
$$\cdot; \mathit{sbnds} : \mathit{sdecs}; S$$
with no imports to a TSIL expression
$$[\mathit{sbnds}, \mathit{lab} \triangleright \mathit{var}{=}\{\}].\mathit{lab} : \{\}$$
where lab and 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.

Notation. We write $\mathit{decs}, \mathit{sdecs}$ to extend a context decs, implicitly dropping the labels in sdecs. We define the domain of a structure declaration list, $\mathrm{dom}(\mathit{sdecs})$, by
$$\mathrm{dom}(\mathit{lab}_1 \triangleright \mathit{dec}_1, \ldots, \mathit{lab}_n \triangleright \mathit{dec}_n) = \{\mathit{lab}_1, \ldots, \mathit{lab}_n\}.$$
We write $\{\mathit{var}/\mathit{var}'\}L$ for the capture-free substitution of var for free occurrences of var' in L.[4]

Linkset merge. The rules for linkset merge $\mathit{decs} \vdash L_1{+\!\!+}L_2 \rightsquigarrow L_3$ combine L_1 and L_2 to produce L_3. The rules presuppose that L_1 is well-formed with respect to decs but permit L_2 to make reference (via free TSIL variables) not only to decs but to the

[4] Linkset bound variables and scopes are discussed in Appendix A.

imports and exports of L_1. Formally, the rules satisfy the following property.[5]

If $L_1 = \mathit{sdecs}_1 \to \mathit{sbnds} : \mathit{sdecs}$
and $\mathit{decs} \vdash L_1\ \mathsf{ok}$
and $\mathit{decs}, \mathit{sdecs}_1, \mathit{sdecs} \vdash L_2\ \mathsf{ok}$,
and $\mathit{decs} \vdash L_1 \mathbin{+\!\!+} L_2 \leadsto L_3$,
then $\mathit{decs} \vdash L_3\ \mathsf{ok}$.

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 L_2 from left to right. If L_2 has no imports, then the following rule applies.

$$\frac{L = \mathit{sdecs}_0 \to \mathit{sbnds} : \mathit{sdecs}}{\begin{array}{l}\mathit{decs} \vdash L \mathbin{+\!\!+} (\cdot \to \mathit{sbnds}' : \mathit{sdecs}') \leadsto \\ \quad \mathit{sdecs}_0 \to \mathit{sbnds} \mathbin{+\!\!+} \mathit{sbnds}' : \mathit{sdecs} \mathbin{+\!\!+} \mathit{sdecs}'\end{array}}$$

L_3 imports what L_1 does and exports what L_1 and L_2 do. To ensure that L_3 is well-formed, the rule uses the TS function $\mathbin{+\!\!+}$ to concatenate the exports in L_1 with the exports in L_2, renaming labels exported by L_1 that are also exported by L_2.

Otherwise, the rules examine the first import $\mathit{lab} \triangleright \mathit{var}{:}\mathit{sig}$ in L_2 and distinguish three mutually exclusive cases:

- L_1 exports lab.

$$\frac{\begin{array}{c}\mathit{sdecs} = \mathit{sdecs}'', \mathit{lab} \triangleright \mathit{var}'{:}\mathit{sig}', \mathit{sdecs}''' \\ \mathit{decs}, \mathit{sdecs}_0, \mathit{sdecs} \vdash_{\mathsf{sub}} \mathit{var}'{:}\mathit{sig}' \preceq \mathit{sig} \leadsto \mathit{mod}{:}\mathit{sig}'' \\ \mathit{sbnd} := \mathit{lab} \triangleright \mathit{var}{=}\mathit{mod} \quad \mathit{sdec} := \mathit{lab} \triangleright \mathit{var}{:}\mathit{sig}'' \\ L := \mathit{sdecs}_0 \to \mathit{sbnds} \mathbin{+\!\!+} \mathit{sbnd} : \mathit{sdecs} \mathbin{+\!\!+} \mathit{sdec} \\ \mathit{decs} \vdash L \mathbin{+\!\!+} (\mathit{sdecs}_1 \to \mathit{sbnds}' : \mathit{sdecs}') \leadsto L''\end{array}}{\begin{array}{l}\mathit{decs} \vdash (\mathit{sdecs}_0 \to \mathit{sbnds} : \mathit{sdecs}) \mathbin{+\!\!+} \\ \quad (\mathit{lab} \triangleright \mathit{var}{:}\mathit{sig}, \mathit{sdecs}_1 \to \mathit{sbnds}' : \mathit{sdecs}') \leadsto L''\end{array}}$$

  The first premise picks out the L_1 export $\mathit{lab} \triangleright \mathit{var}'{:}\mathit{sig}'$ 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 $\mathit{var}'{:}\mathit{sig}'$ 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 $\mathit{sbnd} : \mathit{sdec}$ 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 $\mathbin{+\!\!+}$ to construct its exports.

- L_1 imports lab but does not export it.

$$\frac{\begin{array}{c}\mathit{lab} \notin \mathrm{dom}(\mathit{sdecs}) \\ \mathit{sdecs}_0 = \mathit{sdecs}'', \mathit{lab} \triangleright \mathit{var}'{:}\mathit{sig}', \mathit{sdecs}''' \\ \mathit{decs}, \mathit{sdecs}_0, \mathit{sdecs} \vdash \mathit{sig} \equiv \mathit{sig}' : \mathsf{Sig} \\ L' := \{\mathit{var}'/\mathit{var}\}(\mathit{sdecs}_1 \to \mathit{sbnds}' : \mathit{sdecs}') \\ \mathit{decs} \vdash (\mathit{sdecs}_0 \to \mathit{sbnds} : \mathit{sdecs}) \mathbin{+\!\!+} L' \leadsto L''\end{array}}{\begin{array}{l}\mathit{decs} \vdash (\mathit{sdecs}_0 \to \mathit{sbnds} : \mathit{sdecs}) \mathbin{+\!\!+} \\ \quad (\mathit{lab} \triangleright \mathit{var}{:}\mathit{sig}, \mathit{sdecs}_1 \to \mathit{sbnds}' : \mathit{sdecs}') \leadsto L''\end{array}}$$

  The first premise ensures L_1 does not export lab. The second premise picks out the L_1 import $\mathit{lab} \triangleright \mathit{var}'{:}\mathit{sig}'$. 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.

[5] In this description of linkset merge, we suppress all details related to signature abbreviations.

Judgement...	*Meaning...*
$\mathit{project} \leadsto L$	project elaboration
$\Gamma \vdash \mathit{srcunit} \leadsto L$	unit elaboration
$\Gamma \vdash \mathit{topdec} \leadsto L$	top-level declaration elaboration
$\Gamma \vdash \mathit{impexp} \leadsto L$	import expression elaboration
$\Gamma \vdash \mathit{sigbind} \leadsto S$	signature binding elaboration
$\Gamma \vdash_{\mathsf{ctx}} \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}$	signature lookup
$\Gamma \vdash_{\mathsf{ctx}} \mathit{unitid} \leadsto S$	
$\Gamma\ \mathsf{ok}$	Γ is well-formed

Figure 6. Elaboration judgements

$\mathit{project}$	::=	$\cdot$	empty
		$\mathit{project}, \mathit{srcunit}$	source unit
		$\mathit{project}, L$	compiled unit(s)
$\mathit{srcunit}$	::=	`unit` unitid = topdec	unit declaration
topdec	::=	`import` impexp	open units
		strdec	
		`signature` $\mathit{sigbind}$	
		`local` topdec_1 `in` topdec_2 `end`	
		$\mathit{topdec}_1\ \mathit{topdec}_2$	
impexp	::=	unitid ⟨`intf` spec `end`⟩	open unitid
		$\mathit{impexp}_1\ \mathit{impexp}_2$	
$\mathit{sigbind}$	::=	$\mathit{sigid} = \mathit{sigexp}$ ⟨`and` $\mathit{sigbind}$⟩	

Figure 7. SMLSC abstract syntax

- L_1 neither imports nor exports lab.

$$\frac{\begin{array}{c}\mathit{lab} \notin \mathrm{dom}(\mathit{sdecs}) \cup \mathrm{dom}(\mathit{sdecs}_0) \\ \mathit{decs}, \mathit{sdecs}_0, \mathit{sdecs} \vdash \mathit{sig} \equiv \mathit{sig}' : \mathsf{Sig} \\ \mathit{decs}, \mathit{sdecs}_0 \vdash \mathit{sig}' : \mathsf{Sig} \\ L := \mathit{sdecs}_0, \mathit{lab} \triangleright \mathit{var}{:}\mathit{sig}' \to \mathit{sbnds} : \mathit{sdecs} \\ \mathit{decs} \vdash L \mathbin{+\!\!+} (\mathit{sdecs}_1 \to \mathit{sbnds}' : \mathit{sdecs}') \leadsto L''\end{array}}{\begin{array}{l}\mathit{decs} \vdash (\mathit{sdecs}_0 \to \mathit{sbnds} : \mathit{sdecs}) \mathbin{+\!\!+} \\ \quad (\mathit{lab} \triangleright \mathit{var}{:}\mathit{sig}, \mathit{sdecs}_1 \to \mathit{sbnds}' : \mathit{sdecs}') \leadsto L''\end{array}}$$

  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 given 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.

- $\mathit{project} \leadsto L$. Elaborate $\mathit{project}$, using linkset merge to accumulate a resulting linkset L. A source unit is elaborated in a context Γ that declares the imports and exports in L.
- $\Gamma \vdash \mathit{srcunit} \leadsto L$. Elaborate the topdec in $\mathit{srcunit}$ to the linkset

$$\mathit{sdecs}_0 \to \mathit{sbnds} : \mathit{sdecs}; S.$$

  The imports sdecs_0 arise from the import declarations in topdec. The exports $\mathit{sbnds} : \mathit{sdecs}$ 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

$$\overline{\mathit{unitid}} \triangleright \mathit{var}{=}[\mathit{sbnds}] : \overline{\mathit{unitid}} \triangleright \mathit{var}{:}[\mathit{sdecs}].$$

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

4. 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. The purpose of separate compilation is to permit a client unit to be compiled independently of its implementation. A compiler can exploit this by permitting clients of a separately compiled unit to be compiled in parallel with one another in order to speed up system build times. 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 =
top
  import MatricesImpl :
    intf
      type matrix
      val ** : matrix * matrix -> matrix
      (* ... *)
    end
  infix **
end
```

5. 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 `L1` and `L2` import the same unit `MathLib` at disparate interfaces `I1` and `I2`. This may happen because the developers of `L1` and `L2` compiled using different versions of the handoff unit for `MathLib`, or because the developers wrote their import interfaces by hand. The link

$$\mathit{link}(\mathtt{L1}, \mathtt{L2})$$

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 `MathLib`:

$$\begin{aligned} L1' &= \mathit{link}(\mathtt{MathLib}, \mathtt{L1}) \\ L \quad &= \mathit{link}(L1', \mathtt{L2}) \end{aligned}$$

The first step satisfies the SC import of `MathLib` in `L1`, as long as the actual interface of `MathLib` matches the import interface `I1`. The result $L1'$ does not import `MathLib`, so it does not conflict with the import of `MathLib` in `L2`. $L1'$ does export `MathLib`, so if the actual interface of `MathLib` matches `I2`, then the second link succeeds. Because linking is left-associative, $L = \mathit{link}(\mathtt{MathLib}, \mathtt{L1}, \mathtt{L2})$ accomplishes the same thing.

Any implementation of `MathLib` that satisfies `I1` and `I2` will suffice. Because we do not require unit names to be globally unique, this implementation of `MathLib` might even import `MathLib` (again) and then contain some glue code to make it compatible with the two given interfaces `I1` and `I2` (Figure 8). We expect such cases to be uncommon, the preferred methodology being to use a single handoff unit for all clients.

6. 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 λ-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.

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 λ-calculus $\mathbf{P}_2$ with rank 2 intersection types that has *principal typings*. The principal typings property means that from a term M, one can infer both Γ and τ such that any typing derivation $\Gamma' \vdash M : \tau'$ is an instance of $\Gamma \vdash M : \tau$. In a

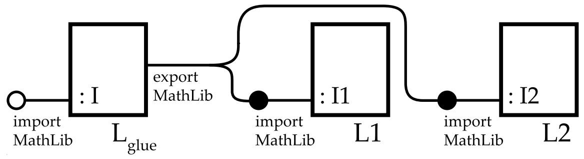

Figure 8. The linkset L_{glue} imports `MathLib` at interface `I` and then exports it to satisfy the imports in `L1` and `L2` at disparate interfaces `I1` and `I2`.

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 4).

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.

7. 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.

References

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

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

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

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

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

[6] Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. Formal specification of the ML Basis system, January 2005. `http://mlton.org/MLBasis`.

[7] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation*. PhD thesis, Department of Computer Science, University of Copenhagen, January 1999.

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

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

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

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

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

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

[14] Trevor Jim. What are principal typings and what are they good for? In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1996.

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

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

[17] MLKit web site. `http://www.itu.dk/research/mlkit/index.php/Main_Page`.

[18] MLton web site. `http://mlton.org/`.

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

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

[21] David Swasey, Tom Murphy, VII, Karl Crary, and Robert Harper. A separate compilation extension to Standard ML (extended technical report). Technical Report CMU-CS-06-133, School of Computer Science, Carnegie Mellon University, 2006.

A. Linking Rules

The typed semantics defines a closed structure $\mathit{mod}_{basis}{:}\mathit{sig}_{basis}$ serving as an initial basis for the TSIL. The elaborator assumes Γ declares $\mathit{basis}{:}\mathit{sig}_{basis}$, 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 $\mathrm{BV}(\mathit{dec})$ for the variable declared by dec, we define the bound variables of a structure declaration list, $\mathrm{BV}(\mathit{sdecs})$, by

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

  A linkset

  $$L = \mathit{sdecs}_0 \rightarrow \mathit{sbnds} : \mathit{sdecs}; S$$

  binds variables $\mathrm{BV}(\mathit{sdecs}_0)$ with scope $\mathit{sbnds} : \mathit{sdecs}; S$ and variables $\mathrm{BV}(\mathit{sdecs})$ with scope S. We write $\mathrm{BV}(L)$ for $\mathrm{BV}(\mathit{sdecs}_0) \cup \mathrm{BV}(\mathit{sdecs})$.
- 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 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 $\mathit{sigid}{=}\mathit{sig}, S$ to match the first binding in the list.
  - We usually omit the initial $\cdot$; for example,

    $$\mathit{sigid}_1{=}\mathit{sig}_1, \ldots, \mathit{sigid}_1{=}\mathit{sig}_1.$$
- We define the domain of a signature abbreviation, $\mathrm{dom}(S)$, by

  $$\begin{array}{lcl} \mathrm{dom}(\cdot) & = & \emptyset \\ \mathrm{dom}(S, \mathit{sigid}{=}\mathit{sig}) & = & \mathrm{dom}(S) \cup \{\mathit{sigid}\} \\ \mathrm{dom}(S, \mathit{unitid}{=}S_u) & = & \mathrm{dom}(S) \cup \{\mathit{unitid}\}. \end{array}$$
- We define the function $S{+\!\!+}S'$ by

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

  It concatenates S and S', making the result well-formed by dropping signature abbreviations if $\mathrm{dom}(S) \cap \mathrm{dom}(S') \neq \emptyset$.
- We write both "=" and ":=" in side-conditions. Interpreting the rules algorithmically, the former pattern-matches inputs, and the latter specifies an output.

$\boxed{\mathit{decs} \vdash L\ \mathsf{ok}}$

$$\frac{\begin{array}{c} \mathit{decs} \vdash \mathit{sdecs}_0\ \mathsf{ok} \\ \mathit{decs}, \mathit{sdecs}_0 \vdash \mathit{sbnds} : \mathit{sdecs} \\ \mathit{decs}, \mathit{sdecs}_0, \mathit{sdecs} \vdash S\ \mathsf{ok} \\ \mathit{sdecs}_0 = \mathit{lab}_1 \triangleright \mathit{var}_1{:}[\mathit{sdecs}_1], \ldots, \mathit{lab}_n \triangleright \mathit{var}_n{:}[\mathit{sdecs}_n] \end{array}}{\mathit{decs} \vdash \mathit{sdecs}_0 \rightarrow \mathit{sbnds} : \mathit{sdecs}; S\ \mathsf{ok}} \quad (1)$$

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

$\boxed{\mathit{decs} \vdash S\ \mathsf{ok}}$

$$\frac{\vdash \mathit{decs}\ \mathsf{ok}}{\mathit{decs} \vdash \cdot\ \mathsf{ok}} \quad (2)$$

$$\frac{\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig} \quad \mathit{decs} \vdash S\ \mathsf{ok} \quad \mathit{sigid} \notin \mathrm{dom}(S)}{\mathit{decs} \vdash \mathit{sigid}{=}\mathit{sig}, S\ \mathsf{ok}} \quad (3)$$

$$\frac{\begin{array}{c} \mathit{decs} \vdash S'\ \mathsf{ok} \quad \mathit{decs} \vdash S\ \mathsf{ok} \quad \mathit{unitid} \notin \mathrm{dom}(S) \\ S' = (\mathit{sigid}_1{=}\mathit{sig}_1, \ldots, \mathit{sigid}_n{=}\mathit{sig}_n) \end{array}}{\mathit{decs} \vdash \mathit{unitid}{=}S', S\ \mathsf{ok}} \quad (4)$$

$\boxed{L \rightsquigarrow \mathit{exp} : \{\}}$

$$\frac{\mathit{lab} \notin \mathrm{dom}(\mathit{sdecs})}{\cdot \rightarrow \mathit{sbnds} : \mathit{sdecs}; S \rightsquigarrow [\mathit{sbnds}, \mathit{lab}{=}\{\}].\mathit{lab} : \{\}} \quad (5)$$

$$\frac{\begin{array}{c} L_{basis} := \cdot \rightarrow \mathsf{basis}{=}\mathit{mod}_{basis} : \mathsf{basis}{:}\mathit{sig}_{basis}; \cdot \\ \vdash L_{basis}{+\!\!+}L \rightsquigarrow L' \quad L' \rightsquigarrow \mathit{exp} : \{\} \end{array}}{L \rightsquigarrow \mathit{exp} : \{\}} \quad (6)$$

$\boxed{\mathit{decs} \vdash L{+\!\!+}L' \rightsquigarrow L''}$

$$\frac{L = \mathit{sdecs}_0 \rightarrow \mathit{sbnds} : \mathit{sdecs}; S}{\begin{array}{l} \mathit{decs} \vdash L{+\!\!+}(\cdot \rightarrow \mathit{sbnds}' : \mathit{sdecs}'; S') \rightsquigarrow \\ \quad \mathit{sdecs}_0 \rightarrow \mathit{sbnds}{+\!\!+}\mathit{sbnds}' : \mathit{sdecs}{+\!\!+}\mathit{sdecs}'; S{+\!\!+}S' \end{array}} \quad (7)$$

$$\frac{\begin{array}{c} sdecs = sdecs'', lab \triangleright var' {:} sig', sdecs''' \\ decs, sdecs_0, sdecs \vdash_{\mathsf{sub}} var' {:} sig' \preceq sig \leadsto mod {:} sig'' \\ sbnd := lab \triangleright var = mod \quad sdec := lab \triangleright var {:} sig'' \\ L := sdecs_0 \to sbnds +\!\!+ sbnd : sdecs +\!\!+ sdec; S \\ decs \vdash L +\!\!+ (sdecs_1 \to sbnds' : sdecs'; S') \leadsto L'' \end{array}}{\begin{array}{l} decs \vdash (sdecs_0 \to sbnds : sdecs; S) +\!\!+ \\ \quad (lab \triangleright var {:} sig, sdecs_1 \to sbnds' : sdecs'; S') \leadsto L'' \end{array}} \quad (8)$$

$$\frac{\begin{array}{c} lab \notin \mathrm{dom}(sdecs) \\ sdecs_0 = sdecs'', lab \triangleright var' {:} sig', sdecs''' \\ decs, sdecs_0, sdecs \vdash sig \equiv sig' : \mathsf{Sig} \\ L' := \{var'/var\}(sdecs_1 \to sbnds' : sdecs'; S') \\ decs \vdash (sdecs_0 \to sbnds : sdecs; S) +\!\!+ L' \leadsto L'' \end{array}}{\begin{array}{l} decs \vdash (sdecs_0 \to sbnds : sdecs; S) +\!\!+ \\ \quad (lab \triangleright var {:} sig, sdecs_1 \to sbnds' : sdecs'; S') \leadsto L'' \end{array}} \quad (9)$$

$$\frac{\begin{array}{c} lab \notin \mathrm{dom}(sdecs) \cup \mathrm{dom}(sdecs_0) \\ decs, sdecs_0, sdecs \vdash sig \equiv sig' : \mathsf{Sig} \\ decs, sdecs_0 \vdash sig' : \mathsf{Sig} \\ L := sdecs_0, lab \triangleright var {:} sig' \to sbnds : sdecs; S \\ decs \vdash L +\!\!+ (sdecs_1 \to sbnds' : sdecs'; S') \leadsto L'' \end{array}}{\begin{array}{l} decs \vdash (sdecs_0 \to sbnds : sdecs; S) +\!\!+ \\ \quad (lab \triangleright var {:} sig, sdecs_1 \to sbnds' : sdecs'; S') \leadsto L'' \end{array}} \quad (10)$$

B. 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 $sdec, sdecs; S$ binds the variable $\mathrm{BV}(sdec)$ with scope $sdecs; S$ and a context $sdecs; S$ binds variables $\mathrm{BV}(sdecs)$ with scope S. We define $\mathrm{BV}(\Gamma)$ by $\mathrm{BV}(sdecs)$. Second, we extend the syntax for TSEL signature expressions:

$$\begin{array}{lll} sigexp & ::= & \ldots \\ & & sigid \quad \text{signature identifier} \end{array}$$

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

$$\frac{\Gamma \vdash_{\mathsf{ctx}} sigid \leadsto sig : \mathsf{Sig}}{\Gamma \vdash sigid \leadsto sig : \mathsf{Sig}}$$

to elaborate signature identifiers.

We use the following definitions and notation.

- To extend an elaboration context $\Gamma = sdecs; S$, we write

  Γ, dec for $sdecs, 1 \triangleright dec; S$,
  $\Gamma, sdecs'$ for $sdecs, sdecs'; S$, and
  Γ, S' for $sdecs; S, S'$.

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

  $$R(lab_1 \triangleright dec_1, \ldots, lab_n \triangleright dec_n) \ = \ 1 \triangleright dec_1, \ldots, 1 \triangleright dec_n.$$

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

  $$U(lab_1 \triangleright dec_1, \ldots, lab_n \triangleright dec_n) = dec_1, \ldots, dec_n.$$

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

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

  $$\begin{array}{l} \sigma(var, \cdot, S) = S \\ \sigma(var, (lab \triangleright dec, sdecs), S) = \\ \quad \{var.lab/\mathrm{BV}(dec)\}\sigma(var, sdecs, S) \end{array}$$

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

$\boxed{project \leadsto L}$

$$\frac{}{\cdot \leadsto (\cdot \to \cdot : \cdot; \cdot)} \quad (11)$$

$$\frac{\begin{array}{c} project \leadsto L \quad basis \notin \mathrm{BV}(L) \\ L = sdecs_0 \to sbnds : sdecs; S \\ \Gamma := basis {:} sig_{basis}, R(sdecs_0), sdecs; S \\ \Gamma \vdash srcunit \leadsto L' \quad var \notin \mathrm{BV}(L') \\ sdecs'_1 \to sbnds' : sdecs'; S' := \{var/basis\}L' \\ sdecs_1 := \mathsf{basis} \triangleright var {:} sig_{basis}, sdecs'_1 \\ \vdash L +\!\!+ (sdecs_1 \to sbnds' : sdecs'; S') \leadsto L'' \end{array}}{project, srcunit \leadsto L''} \quad (12)$$

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

$$\frac{\begin{array}{c} project \leadsto L \\ \mathrm{BV}(L) \cap \mathrm{BV}(L') = \emptyset \quad \vdash L' \ \mathsf{ok} \quad \vdash L +\!\!+ L' \leadsto L'' \end{array}}{project, L' \leadsto L''} \quad (13)$$

$\boxed{\Gamma \vdash srcunit \leadsto L}$

$$\frac{\begin{array}{c} \Gamma \vdash topdec \leadsto L \\ L = sdecs_0 \to sbnds : sdecs; S \\ var \notin \mathrm{BV}(\Gamma) \cup \mathrm{BV}(L) \\ sbnds' := \overline{unitid} \triangleright var = [sbnds] \\ sdecs' := \overline{unitid} \triangleright var {:} [sdecs] \\ S' := unitid = \sigma(var, sdecs, S) \end{array}}{\begin{array}{l} \Gamma \vdash \mathtt{unit}\ unitid = topdec \leadsto \\ \quad sdecs_0 \to sbnds' : sdecs'; S' \end{array}} \quad (14)$$

$\boxed{\Gamma \vdash topdec \leadsto L}$

$$\frac{\Gamma \vdash impexp \leadsto L}{\Gamma \vdash \mathtt{import}\ impexp \leadsto L} \quad (15)$$

$$\frac{\Gamma \vdash strdec \leadsto sbnds : sdecs}{\Gamma \vdash strdec \leadsto \cdot \to sbnds : sdecs; \cdot} \quad (16)$$

$$\frac{\Gamma \vdash sigbind \leadsto S}{\Gamma \vdash \mathtt{signature}\ sigbind \leadsto \cdot \to \cdot : \cdot; S} \quad (17)$$

$$\frac{\begin{array}{c} \Gamma \vdash topdec \leadsto sdecs_0 \to sbnds : sdecs; S \\ var \notin \mathrm{BV}(\Gamma) \cup \mathrm{BV}(sdecs_0) \\ \Gamma, R(sdecs_0), 1^\star \triangleright var {:} [sdecs], S \vdash topdec' \leadsto L' \\ L := sdecs_0 \to 1 \triangleright var = [sbnds] : 1 \triangleright var {:} [sdecs]; \cdot \\ \Gamma \vdash L +\!\!+ L' \leadsto L'' \end{array}}{\Gamma \vdash \mathtt{local}\ topdec\ \mathtt{in}\ topdec'\ \mathtt{end} \leadsto L''} \quad (18)$$

$$\frac{\begin{array}{c} \Gamma \vdash topdec \leadsto L \\ L = sdecs_0 \to sbnds : sdecs; S \\ \Gamma, R(sdecs_0), sdecs, S \vdash topdec' \leadsto L' \\ \Gamma \vdash L +\!\!+ L' \leadsto L'' \end{array}}{\Gamma \vdash topdec\ topdec' \leadsto L''} \quad (19)$$

$$\boxed{\Gamma \vdash \mathit{impexp} \leadsto L}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ctx}} \overline{\mathit{unitid}} \leadsto \mathit{var} : \mathit{sig} \\ \Gamma \vdash_{\mathsf{ctx}} \mathit{unitid} \leadsto S \quad \mathit{var}' \notin \mathrm{BV}(\Gamma) \\ L := \overline{\mathit{unitid}} \triangleright \mathit{var}'{:}\mathit{sig} \rightarrow 1^{\star}{=}\mathit{var}' : 1^{\star}{:}\mathit{sig}; S\end{array}}{\Gamma \vdash \mathit{unitid} \leadsto L} \qquad (20)$$

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

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{spec} \leadsto \mathit{sdecs} \quad \mathit{var}' \notin \mathrm{BV}(\Gamma) \\ \Gamma, \mathit{var}' : [\mathit{sdecs}] \vdash \mathit{var}' : \mathit{sig} \\ L := \overline{\mathit{unitid}} \triangleright \mathit{var}'{:}[\mathit{sdecs}] \rightarrow 1^{\star}{=}\mathit{var}' : 1^{\star}{:}\mathit{sig}; \cdot\end{array}}{\Gamma \vdash \mathit{unitid} : \texttt{intf}\ \mathit{spec}\ \texttt{end} \leadsto L} \qquad (21)$$

Rule 21: The signature sig should be fully selfified.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{impexp} \leadsto L \quad \Gamma \vdash \mathit{impexp}' \leadsto L' \\ \mathrm{BV}(L) \cap \mathrm{BV}(L') = \emptyset \quad \Gamma \vdash L {+\!\!+} L' \leadsto L''\end{array}}{\Gamma \vdash \mathit{impexp}\ \mathit{impexp}' \leadsto L''} \qquad (22)$$

$$\boxed{\Gamma \vdash \mathit{sigbind} \leadsto S}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \quad S \ := \ \mathit{sigid}{=}\mathit{sig} \\ \langle \Gamma \vdash \mathit{sigbind} \leadsto S' \quad \mathit{sigid} \notin \mathrm{dom}(S') \rangle\end{array}}{\Gamma \vdash \mathit{sigid} = \mathit{sigexp}\ \langle \texttt{and}\ \mathit{sigbind} \rangle \leadsto S \langle, S' \rangle} \qquad (23)$$

Rule 23: Either all optional elements or none must be present.

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

$$\frac{}{\mathit{sdecs}; S, \mathit{sigid}{=}\mathit{sig} \vdash \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}} \qquad (24)$$

$$\frac{\begin{array}{c}\mathit{sigid}' \neq \mathit{sigid} \\ \mathit{sdecs}; S \vdash \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}\end{array}}{\mathit{sdecs}; S, \mathit{sigid}'{=}\mathit{sig}' \vdash \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}} \qquad (25)$$

$$\frac{\mathit{sdecs}; S \vdash \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}}{\mathit{sdecs}; S, \mathit{unitid}{=}S' \vdash \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}} \qquad (26)$$

$$\boxed{\Gamma \vdash_{\mathsf{ctx}} \mathit{unitid} \leadsto S}$$

$$\frac{}{\mathit{sdecs}; S, \mathit{unitid}{=}S' \vdash \mathit{unitid} \leadsto S'} \qquad (27)$$

$$\frac{\begin{array}{c}\mathit{unitid}' \neq \mathit{unitid} \\ \mathit{sdecs}; S \vdash \mathit{unitid} \leadsto S''\end{array}}{\mathit{sdecs}; S, \mathit{unitid}'{=}S' \vdash \mathit{unitid} \leadsto S''} \qquad (28)$$

$$\frac{\mathit{sdecs}; S \vdash \mathit{unitid} \leadsto S'}{\mathit{sdecs}; S, \mathit{sigid}{=}\mathit{sig} \vdash \mathit{unitid} \leadsto S'} \qquad (29)$$

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

$$\frac{\vdash U(\mathit{sdecs})\ \mathsf{ok}}{\mathit{sdecs}; \cdot\ \mathsf{ok}} \qquad (30)$$

$$\frac{\mathit{sdecs}; S\ \mathsf{ok} \quad \mathit{sdecs} \vdash \mathit{sig} : \mathsf{Sig}}{\mathit{sdecs}; S, \mathit{sigid}{=}\mathit{sig}\ \mathsf{ok}} \qquad (31)$$

$$\frac{\begin{array}{c}\mathit{sdecs}; S\ \mathsf{ok} \quad \mathit{sdecs} \vdash S'\ \mathsf{ok} \\ S' = (\mathit{sigid}_1{=}\mathit{sig}_1, \ldots, \mathit{sigid}_n{=}\mathit{sig}_n)\end{array}}{\mathit{sdecs}; S, \mathit{unitid}{=}S'\ \mathsf{ok}} \qquad (32)$$