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

David Swasey   Tom Murphy VII   Karl Crary
Robert Harper

September 6, 2006
CMU-CS-06-133

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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.

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

This report supersedes CMU-CS-06-104.

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.[1] 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, permiting 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

srcunit	::=	`unit` *unitid* = `unit` *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* ⟨: *intexp*⟩	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′* ⟨`and` *funspec*⟩	

Figure 1: SMLSC concrete syntax

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 *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:

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

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 =
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
```

```
    import CollectionsImpl :
      intf
        structure Queue : QUEUE
      end
  end
```

The implementation of the collections library is imported from the unit `CollectionsImpl`.[2] 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:

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

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 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 λ-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 *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 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 $decs = dec_1, \ldots, dec_n$ 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.

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,

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

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

Figure 2: TSIL judgements (summary)

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)

- $\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig}$. The signature *sig* is well-formed.

- $\mathit{decs} \vdash \mathit{sig} \equiv \mathit{sig}' : \mathsf{Sig}$. The signatures *sig* and sig' declare the same components, in the same order, with the same labels, and corresponding type components are equivalent.

- $\mathit{decs} \vdash \mathit{sbnds} : \mathit{sdecs}$. 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

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

  holds if $\mathit{decs} \vdash \mathit{mod} : \mathit{sig}$ and $\mathit{decs}, \mathit{var}{:}\mathit{sig} \vdash \mathit{sbnds} : \mathit{sdecs}$.

- $\mathit{decs} \vdash \mathit{mod} : \mathit{sig}$. 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.

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

- $\Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig}$. 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.

- $\Gamma \vdash \mathit{spec} \rightsquigarrow \mathit{sdecs}$. 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

$$\begin{array}{rcl} \mathit{sbnds} : \mathit{sdecs} & = & \overline{T} \triangleright t_1{=}\{\} : \overline{T} \triangleright t_1{:}\Omega{=}\{\} \\ \mathit{sbnds}' : \mathit{sdecs}' & = & \overline{T} \triangleright t_2{=}\mathtt{Int} : \overline{T} \triangleright t_2{:}\Omega{=}\mathtt{Int}, \end{array}$$

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.

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

Figure 4: Linking judgements

$$
\begin{array}{lcll}
L & ::= & sdecs_0 \rightarrow sbnds : sdecs; S & \text{linkset} \\
S & ::= & \cdot & \\
 & & S, sigid = sig & \text{top-level} \\
 & & S, unitid = S' & \text{declared by } unitid
\end{array}
$$

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

  $$
  \begin{array}{rl}
  sdecs_{AB} = & A \triangleright a{:}[T \triangleright t{:}\Omega, X \triangleright x{:}t], \\
   & B \triangleright b{:}[Y \triangleright y{:}a.T]
  \end{array}
  $$

  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 $sbnds : sdecs$ are the TSIL code associated with the linkset. They may make reference to the linkset's imports. Continuing our example, the exports

  $$
  \begin{array}{rl}
  sbnds_{ZR} : sdecs_{ZR} = & Z \triangleright z{=}b.Y, R \triangleright r{=}a.T : \\
   & Z \triangleright z{:}a.T, R \triangleright r{:}\Omega{=}a.T
  \end{array}
  $$

  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

  $$S_{\texttt{SIG}} \quad = \quad \texttt{SIG}{=}[M \triangleright m{:}\Omega{=}r]$$

  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 $L \rightsquigarrow exp : \{\}$ translates a linkset

$$\cdot \rightarrow sbnds : sdecs; S$$

with no imports to a TSIL expression

$$[sbnds, lab \triangleright var{=}\{\}].lab : \{\}$$

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

$$\mathrm{dom}(lab_1 \triangleright dec_1, \ldots, lab_n \triangleright dec_n) = \{lab_1, \ldots, lab_n\}.$$

We write $\{var/var'\}L$ for the capture-free substitution of var for free occurrences of var' in L.[3]

Linkset merge. The rules for linkset merge $decs \vdash L_1 {+\!\!+} L_2 \rightsquigarrow L_3$ 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 imports and exports of L_1. Formally, the rules satisfy the following property.[4]

> If $L_1 = sdecs_1 \rightarrow sbnds : sdecs$
> and $decs \vdash L_1$ ok
> and $decs, sdecs_1, sdecs \vdash L_2$ ok,
> and $decs \vdash L_1 {+\!\!+} L_2 \rightsquigarrow L_3$,
> then $decs \vdash L_3$ 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.

$$\frac{L = sdecs_0 \rightarrow sbnds : sdecs}{\begin{array}{l} decs \vdash L {+\!\!+} (\cdot \rightarrow sbnds' : sdecs') \rightsquigarrow \\ \quad sdecs_0 \rightarrow sbnds {+\!\!+} sbnds' : sdecs {+\!\!+} sdecs' \end{array}}$$

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 ${+\!\!+}$ 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 $lab \triangleright var{:}sig$ in L_2 and distinguish three mutually exclusive cases:

- L_1 exports lab.

$$\frac{\begin{array}{c} sdecs = sdecs'', lab \triangleright var'{:}sig', sdecs''' \\ decs, sdecs_0, sdecs \vdash_{\mathsf{sub}} var'{:}sig' \preceq sig \rightsquigarrow mod{:}sig'' \\ sbnd := 1 \triangleright var{=}mod \quad sdec := 1 \triangleright var{:}sig'' \\ L := sdecs_0 \rightarrow sbnds {+\!\!+} sbnd : sdecs {+\!\!+} sdec \\ decs \vdash L {+\!\!+} (sdecs_1 \rightarrow sbnds' : sdecs') \rightsquigarrow L'' \end{array}}{\begin{array}{l} decs \vdash (sdecs_0 \rightarrow sbnds : sdecs) {+\!\!+} \\ \quad (lab \triangleright var{:}sig, sdecs_1 \rightarrow sbnds' : sdecs') \rightsquigarrow L'' \end{array}}$$

[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.

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

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.

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

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 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.

- $project \leadsto L$. Elaborate $project$, 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.

- $\Gamma \vdash srcunit \leadsto L$. Elaborate the $topdec$ in $srcunit$ to the linkset

$$sdecs_0 \to sbnds : sdecs; S.$$

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

Figure 6: Elaboration judgements

project	::=	$\cdot$	empty
		$project, srcunit$	source unit
		$project, L$	compiled unit(s)
srcunit	::=	$\texttt{unit}\ unitid = topdec$	unit declaration
topdec	::=	$\texttt{import}\ impexp$	open units
		$strdec$	
		$\texttt{signature}\ sigbind$	
		$\texttt{local}\ topdec_1\ \texttt{in}\ topdec_2\ \texttt{end}$	
		$topdec_1\ topdec_2$	
impexp	::=	$unitid\ \langle \texttt{:}\ \texttt{intf}\ spec\ \texttt{end}\rangle$	open *unitid*
		$impexp_1\ impexp_2$	
sigbind	::=	$sigid = sigexp\ \langle \texttt{and}\ sigbind\rangle$	

Figure 7: SMLSC abstract syntax

The imports $sdecs_0$ arise from the import declarations in *topdec*. The exports $sbnds : sdecs$ 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

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

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

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

$$\begin{array}{lcll} udecs & ::= & \cdot & \text{empty} \\ & & udecs, srcunit & \text{unit declaration} \end{array}$$

Figure 8: Unit syntax

Judgement...	*Meaning...*
$B \vdash \mathit{strdec} \Rightarrow E$	structure declaration elaboration
$B \vdash \mathit{sigdec} \Rightarrow G$	signature declaration elaboration
$B \vdash \mathit{fundec} \Rightarrow F$	functor declaration elaboration
$B \vdash \mathit{sigexp} \Rightarrow \Sigma$	signature expression elaboration
$B \vdash \mathit{spec} \Rightarrow E$	specification elaboration
$\Sigma \geq E$ using φ	signature instantiation
$E_1 \succ E_2$	enrichment

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

- $B \vdash \mathit{spec} \Rightarrow E$. The specification *spec* is well-formed and specifies the components in E.
- $\Sigma \geq E$ using φ. The environment E is an instance of the signature Σ using the realization φ.
- $E_1 \succ E_2$. 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

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

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 *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 *strexp* `:>` *sigexp* generates types after elaborating *strexp* and *sigexp*. Consider the following structure declaration.

```
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 `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 "`struct type a = int type b = a end`" elaborates to an environment E mapping type constructors `a` and `b` to types stamped with $t_{\texttt{int}}$ (where $t_{\texttt{int}}$ is the type name associated with `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.

$$
\begin{array}{lcl}
\mathit{uspecs} & \in & \text{UnitSpecs} = \bigcup_{n\geq 0} \text{UnitSpecs}^n \\
\mathit{unitid}{:}\Upsilon_1, \ldots, \mathit{unitid}{:}\Upsilon_n & \in & \text{UnitSpecs}^n = (\text{UnitId} \times \text{UnitSig})^n \\
\Upsilon \text{ or } (T)(F, G, E) & \in & \text{UnitSig} = \text{TyNameSet} \times (\text{FunEnv} \times \text{SigEnv} \times \text{Env}) \\
\Gamma \text{ or } B, U & \in & \text{UnitBasis} = \text{Basis} \times \text{UnitEnv} \\
U & \in & \text{UnitEnv} = \text{UnitId} \stackrel{\text{fin}}{\to} \text{Basis} \\
\mathit{unitid} & \in & \text{UnitId} \quad \text{(unit identifiers)}
\end{array}
$$

Figure 10: Static TDIL for unit declaration lists

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

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 *strexp* `:` *sigexp* induces sharing after elaborating *strexp* and *sigexp*. Consider the following structure declaration.

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

Types `B.b` and `int` share. To induce sharing, TD uses capture-avoiding substitution from type names to types: The rule chooses and applies a *realization* φ. 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 $t_{\texttt{int}}$ (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 $t_{\texttt{int}}$ 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.
- $A \stackrel{\text{fin}}{\to} B$ denotes the set of partial functions from A to B with finite domain.

Judgement...	*Meaning...*
$\Gamma \vdash \mathit{udecs} : \mathit{uspecs}$	*udecs* has specification list *uspecs*
$\Gamma \vdash \mathit{topdec} : B$	*topdec* has basis B
$\Gamma \vdash \mathit{impexp} : F, G, E$	*impexp* has components F, G, E
$B \vdash \mathit{intexp} \Rightarrow \Upsilon$	interface expression elaboration
$B \vdash \mathit{topspec} \Rightarrow F, E$	top-level specification elaboration
$B \vdash \mathit{funspec} \Rightarrow F$	functor specification elaboration
$\Gamma\ \mathsf{ok}$	Γ is well-formed
$\Gamma \vdash \Upsilon : \mathsf{Sig}$	Υ is well-formed
$\Gamma \vdash \mathit{uspecs}\ \mathsf{ok}$	*uspecs* is well-formed
$E : \Sigma\ \mathsf{using}\ \varphi$	signature matching
$\Sigma \equiv \Sigma'\ \mathsf{using}\ \varphi$	signature equivalence
$\Upsilon \equiv \Upsilon'\ \mathsf{using}\ \varphi$	interface equivalence
$F, G, E : \Upsilon\ \mathsf{using}\ \varphi$	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. Υ specifies the components in environments F, G, and E, binding type names T with scope (F, G, E).

A unit specification list $\mathit{unitid}_1{:}\Upsilon_1, \ldots, \mathit{unitid}_n{:}\Upsilon_n$ describes a unit declaration list. Writing $\mathrm{BT}(\Upsilon)$ for the type names bound in Υ, a unit specification list binds $\mathrm{BT}(\Upsilon_i)$ with scope $\mathit{unitid}_{i+1}{:}\Upsilon_{i+1}, \ldots, \mathit{unitid}_n{:}\Upsilon_n$ 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 $U(\mathit{unitid}) = T', F', G', E'$, then $T' \subset T$ records the type names generated by *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.

- $\Gamma \vdash \mathit{udecs} : \mathit{uspecs}$. 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

$$\Gamma \vdash (\texttt{unit}\ \mathit{unitid} = \texttt{unit}\ \mathit{topdec}\ \texttt{end}, \mathit{udecs}) : (\mathit{unitid}{:}(T)(F, G, E), \mathit{uspecs})$$

  holds if $\Gamma \vdash \mathit{topdec} : T, F, G, E$ and $\Gamma + T + \{\mathit{unitid} \mapsto T, F, G, E\} \vdash \mathit{udecs} : \mathit{uspecs}$.[5]

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

- $\Gamma \vdash \mathit{impexp} : F, G, E$. The import expression *impexp* imports the components in F, G, E from Γ.

[5] The notation $\Gamma + T + \{\mathit{unitid} \mapsto T, F, G, E\}$ extends the state then the context in Γ (see Appendix C).

- $B \vdash intexp \Rightarrow \Upsilon$. The interface expression *intexp* specifies the components in Υ. Since interface expressions do not describe signature declarations, the signature environment G in Υ must be empty.
- $B \vdash topspec \Rightarrow F, E$. The top-level specification *topspec* specifies the components in F, E. No identifier may be specified twice.
- $B \vdash funspec \Rightarrow F$. The functor specification *funspec* specifies the components in F. No identifier may be specified twice.
- Γ ok. The unit basis $\Gamma = (T, F, G, E), U$ is well-formed: T contains the free type names in F, G, E, U.
- $\Gamma \vdash \Upsilon : \mathsf{Sig}$. The interface $\Upsilon = (T)(F, G, E)$ is well-formed: No type name $t \in T$ occurs in Γ's state T' and $T \cup T'$ contains the free type names in F, G, E.
- $\Gamma \vdash uspecs$ ok. The unit specification list *uspecs* is well-formed. For example, the judgement

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

  holds if $\Gamma \vdash \Upsilon : \mathsf{Sig}$ and $\Gamma + \mathrm{BT}(\Upsilon) \vdash uspecs$ ok.
- $E : \Sigma$ using φ. The environment E matches the signature Σ using the realization φ.
- $\Sigma \equiv \Sigma'$ using φ. The signatures $\Sigma = (T)E$ and $\Sigma' = (T')E'$ specify the same components and $E = \varphi(E')$.
- $\Upsilon \equiv \Upsilon'$ using φ. The interfaces $\Upsilon = (T)(F, G, E)$ and $\Upsilon' = (T')(F', G', E')$ specify the same components and $F, G, E = \varphi(F', G', E')$.
- $F, G, E : \Upsilon$ using φ. The environments F, G, E match the interface Υ and the realization φ induces the requisite sharing: If the semantic object A refers to type names bound in Υ, 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.

[6] In many cases, static and dynamic TDIL categories have the same names and employ the same metavariables. No confusion can result since the static and dynamic semantics are separate.

Judgement...	*Meaning...*
$s, B \vdash strdec \Rightarrow E, s'$	structure declaration evaluation
$B \vdash fundec \Rightarrow F$	functor declaration evaluation
$IB \vdash sigdec \Rightarrow G$	signature declaration elaboration
$IB \vdash sigexp \Rightarrow I$	signature elaboration
$IB \vdash spec \Rightarrow I$	specification elaboration
$\mathrm{Inter}\ B = IB$	interface basis extraction
$E \downarrow I = E'$	signature ascription

Figure 12: TD's dynamic semantic judgements (summary)

- $s, B \vdash \mathit{strdec} \Rightarrow E/p, s'$. In state s and context B, the structure declaration *strdec* evaluates to state s' and either an environment E or an exception packet p.

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

- $\mathit{IB} \vdash \mathit{sigdec} \Rightarrow G$. The signature declaration *sigdec* declares the signature identifiers in signature environment G.

- $\mathit{IB} \vdash \mathit{sigexp} \Rightarrow I$. The signature expression *sigexp* specifies the components in structure interface I.

- $\mathit{IB} \vdash \mathit{spec} \Rightarrow I$. The specification *spec* specifies the components in I.

- Inter $B = \mathit{IB}$. 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

$$s, B \vdash \mathit{strdec} \Rightarrow E/p, s',$$

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 propogate raised exceptions explicitly, referring to them as exception packets p. (Compound metavariables like E/p range over the disjoint union of two TDIL categories.)

$$
\begin{array}{rcl}
\mathit{UI} \text{ or } \mathit{FI}, I & \in & \mathrm{UnitInt} = \mathrm{FunIntEnv} \times \mathrm{Int} \\
\mathit{FI} & \in & \mathrm{FunIntEnv} = \mathrm{FunId} \xrightarrow{\mathrm{fin}} \mathrm{Int} \times \mathrm{Int} \\
\Gamma & \in & \mathrm{UnitBasis} = \mathrm{Basis} \times \mathrm{UnitEnv} \\
U & \in & \mathrm{UnitEnv} = \mathrm{UnitId} \xrightarrow{\mathrm{fin}} \mathrm{Basis}
\end{array}
$$

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* : *intexp* is analagous 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 $\mathit{UI} = \mathit{FI}, I$ 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.

```
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 $\Gamma = B, U$ serves as an evaluation context. The unit environment U is a finite map from unit identifiers to bases: If $U(\mathit{unitid}) = F, G, E$, 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.

- $s, \Gamma \vdash \mathit{udecs} \Rightarrow \Gamma'/p, s'$. Evaluate the unit declaration list *udecs* to a unit basis Γ' or an exception packet p.
- $s, \Gamma \vdash \mathit{topdec} \Rightarrow B/p, s'$. Evaluate the top-level declaration *topdec* to a basis B or an exception packet p.

Judgement...	*Meaning...*
$s, \Gamma \vdash \mathit{udecs} \Rightarrow \Gamma'/p, s'$	unit evaluation
$s, \Gamma \vdash \mathit{topdec} \Rightarrow B/p, s'$	top-level declaration evaluation
$\Gamma \vdash \mathit{impexp} \Rightarrow B$	import expression evaluation
$\mathit{IB} \vdash \mathit{topspec} \Rightarrow \mathit{UI}$	top-level specification elaboration
$\mathit{IB} \vdash \mathit{funspec} \Rightarrow \mathit{FI}$	functor specfication elaboration

Figure 14: Unit evaluation judgements

- $\Gamma \vdash \mathit{impexp} \Rightarrow B$. Evaluate the import expression *impexp* to the basis B.
- $\mathit{IB} \vdash \mathit{topspec} \Rightarrow \mathit{UI}$. The top-level specification *topspec* specifies the components in the unit interface UI.
- $\mathit{IB} \vdash \mathit{funspec} \Rightarrow \mathit{FI}$. The functor specification *funspec* specifies the components in the functor interface environment FI.

4.3 Linking

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

$$\mathit{uspecs}_0 \rightarrow \mathit{exps}$$

comprises imports uspecs_0 and exports *exps*. Exports may take two forms—a unit declaration list $\mathit{udecs} : \mathit{uspecs}$ or a static TDIL basis B.

- The imports 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

  $$\mathit{uspecs}_{AB} = A{:}\Upsilon_A, B{:}\Upsilon_B$$

  express dependency on units A and B.

  Imports specify assumptions to be satisfied by linking. A linkset with imports uspecs_{AB} assumes unit B declares (at least) the components described by the interface Υ_B but can be linked with (a linkset exporting) a unit B providing more components.
- The exports $\mathit{exps} = \mathit{udecs} : \mathit{uspecs}$ are the code associated with the linkset. They may make reference to the linkset's imports (via free unit identifiers and type names).

Judgement...	*Meaning...*
$\Gamma \vdash L\ \mathsf{ok}$	L is well-formed
$\Gamma \vdash \mathit{exps}\ \mathsf{ok}$	*exps* is well-formed
$L \Rightarrow \mathit{udecs}$	L completes to *udecs*
$\Gamma \vdash L{+\!\!+}L' \Rightarrow L''$	L and L' merge to L''
$\Gamma \vdash \mathit{exps}{+\!\!+}\mathit{exps}' \Rightarrow \mathit{exps}''$	*exps* and exps' merge to exps''

Figure 15: Linking judgements

$$
\begin{array}{rcll}
L & ::= & \mathit{uspecs} \to \mathit{exps} & \text{linkset} \\
\mathit{exps} & ::= & \mathit{udecs} : \mathit{uspecs} & \text{units} \\
 & & B & \text{top-level components}
\end{array}
$$

Figure 16: Linkset syntax

- The exports $\mathit{exps} = B$ 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 $L \Rightarrow \mathit{udecs}$ translates a linkset

$$\cdot \to \mathit{udecs} : \mathit{uspecs}$$

with no imports to the unit declaration list *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, $\Gamma + \mathit{uspecs}$, by[8]

$$
\begin{array}{rcl}
\Gamma + \cdot & = & \Gamma \\
\Gamma + (\mathit{unitid}{:}(T)(F,G,E), \mathit{uspecs}) & = & (\Gamma + T + \{\mathit{unitid} \mapsto T, F, G, E\}) + \mathit{uspecs}.
\end{array}
$$

We define the type names bound by a unit specification list, $\mathrm{BT}(\mathit{uspecs})$, by

$$\mathrm{BT}(\mathit{unitid}_1{:}\Upsilon_1, \ldots, \mathit{unitid}_n{:}\Upsilon_n) = \mathrm{BT}(\Upsilon_1) \cup \cdots \cup \mathrm{BT}(\Upsilon_n).$$

We define the domain of a unit specification list, $\mathrm{dom}(\mathit{uspecs})$, by

$$\mathrm{dom}(\mathit{unitid}_1{:}\Upsilon_1, \ldots, \mathit{unitid}_n{:}\Upsilon_n) = \{\mathit{unitid}_1, \ldots, \mathit{unitid}_n\}$$

and the domain of a linkset's exports, $\mathrm{dom}(\mathit{exps})$, by

$$
\begin{array}{rcl}
\mathrm{dom}(\mathit{udecs} : \mathit{uspecs}) & = & \mathrm{dom}(\mathit{uspecs}) \\
\mathrm{dom}(B) & = & \emptyset.
\end{array}
$$

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 Γ but permit L_2 to make reference not only to Γ 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.

$$\frac{\Gamma \vdash \mathit{exps} {+\!\!+} \mathit{exps}' \Rightarrow \mathit{exps}''}{\Gamma \vdash (\mathit{uspecs}_0 \to \mathit{exps}) {+\!\!+} (\cdot \to \mathit{exps}') \Rightarrow \mathit{uspecs}_0 \to \mathit{exps}''}$$

[7] The unit declaration list *udecs* obtained by completion may be evaluated in the initial state s_0 and the unit basis $\Gamma_0 = B_0, \{\}$, 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 $\Gamma(\mathit{unitid})$.

L_3 imports what L_1 does. The rules for $\Gamma \vdash \mathit{exps}{+\!\!+}\mathit{exps}' \Rightarrow \mathit{exps}''$ ensure that L_3 exports what L_1 and L_2 do.

Otherwise, the rules examine the first import $\mathit{unitid}{:}\Upsilon$ in L_2 and distinguish three mutually exclusive cases:

- L_1 exports unitid.

$$\frac{\begin{array}{c}\mathit{unitid} \in \mathrm{dom}(\mathit{uspecs}) \\ (\Gamma + \mathit{uspecs}_0 + \mathit{uspecs})(\mathit{unitid}) = T', F', G', E' \\ F', G', E' : \Upsilon \ \mathsf{using}\ \varphi \\ L' := \varphi(\mathit{uspecs}_1 \to \mathit{udecs}' : \mathit{uspecs}') \\ \Gamma \vdash (\mathit{uspecs}_0 \to \mathit{udecs} : \mathit{uspecs}){+\!\!+}L' \Rightarrow L''\end{array}}{\Gamma \vdash (\mathit{uspecs}_0 \to \mathit{udecs} : \mathit{uspecs}){+\!\!+}(\mathit{unitid}{:}\Upsilon, \mathit{uspecs}_1 \to \mathit{udecs}' : \mathit{uspecs}') \Rightarrow L''}$$

  The first premise ensures L_1 exports unitid. The second premise picks out the L_1 export $\mathit{unitid} : (T')(F', G', E')$ 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 L_2 to the types employed by L_1.

- L_1 imports unitid but does not export it.

$$\frac{\begin{array}{c}\mathit{unitid} \notin \mathrm{dom}(\mathit{exps}) \\ \mathit{uspecs}_0 = \mathit{uspecs}'', \mathit{unitid}{:}\Upsilon', \mathit{uspecs}''' \\ \Upsilon' \equiv \Upsilon \ \mathsf{using}\ \varphi \\ L' := \varphi(\mathit{uspecs}_1 \to \mathit{exps}') \\ \Gamma \vdash (\mathit{uspecs}_0 \to \mathit{exps}){+\!\!+}L' \Rightarrow L''\end{array}}{\Gamma \vdash (\mathit{uspecs}_0 \to \mathit{exps}){+\!\!+}(\mathit{unitid}{:}\Upsilon, \mathit{uspecs}_1 \to \mathit{exps}') \Rightarrow L''}$$

  The first premise ensures L_1 does not export unitid. The second premise picks out the L_1 import $\mathit{unitid}{:}\Upsilon'$; there can be at most once since L_1 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 L_2 to the type names bound by Υ'.

- L_1 neither imports nor exports unitid.

$$\frac{\begin{array}{c}\mathit{unitid} \notin \mathrm{dom}(\mathit{exps}) \cup \mathrm{dom}(\mathit{uspecs}_0) \\ \Upsilon' \equiv \Upsilon \ \mathsf{using}\ \varphi \\ \Gamma + \mathrm{BT}(\mathit{uspecs}_0) \vdash \Upsilon' : \mathsf{Sig} \\ L := \mathit{uspecs}_0, \mathit{unitid}{:}\Upsilon' \to \mathit{exps} \\ L' := \varphi(\mathit{uspecs}_1 \to \mathit{exps}') \\ \Gamma \vdash L{+\!\!+}L' \Rightarrow L''\end{array}}{\Gamma \vdash (\mathit{uspecs}_0 \to \mathit{exps}){+\!\!+}(\mathit{unitid}{:}\Upsilon, \mathit{uspecs}_1 \to \mathit{exps}') \Rightarrow L''}$$

  The first premise ensures that L_1 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 L_1. Linking fails if no such interface exists—when type names exported by L_1 occur in Υ. Otherwise, L is constructed by adding a new import to the imports in L_1 and L' is constructed by changing references to the type names bound in Υ in the remainder of L_2 to the type names bound in Υ'.

Judgement...	*Meaning...*
$project \Rightarrow L$	project elaboration
$\Gamma \vdash srcunit \Rightarrow L$	unit elaboration
$\Gamma \vdash topdec \Rightarrow L$	top-level declaration elaboration
$\Gamma \vdash impexp \Rightarrow L$	import expression elaboration

Figure 17: Elaboration judgements

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.

- $project \Rightarrow L$. 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.
- $\Gamma \vdash srcunit \Rightarrow L$. Elaborate the *topdec* in *srcunit* to the linkset

  $$udecs_0 \to T, F, G, E.$$

  The imports $udecs_0$ 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:

  $$L = udecs_0 \to srcunit : (T)(F, G, E).$$

- $\Gamma \vdash topdec \Rightarrow L$. Elaborate *topdec* using linkset merge.
- $\Gamma \vdash impexp \Rightarrow L$. 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.

$$\begin{array}{lcll} project & ::= & \cdot & \text{empty} \\ & & project, srcunit & \text{source unit} \\ & & project, L & \text{compiled unit(s)} \end{array}$$

Figure 18: SMLSC abstract syntax

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

$$link(L_1, L_2)$$

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

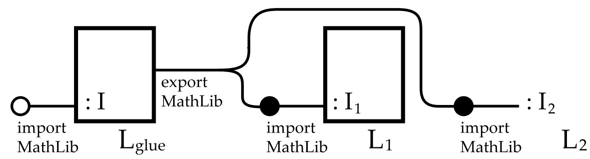

Figure 19: The linkset L_{glue} imports `MathLib` at interface I and then exports it to satisfy the imports in L_1 and L_2 at disparate interfaces I_1 and I_2.

`MathLib`:

$$\begin{aligned} L_1' &= link(MathLib, L_1) \\ L &= link(L_1', L_2) \end{aligned}$$

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

Any implementation of `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 `MathLib` might even import `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 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.

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 accross `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 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 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

[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`.

A TS Linking Rules

The typed semantics defines a closed structure $\mathit{mod}_{\mathit{basis}}{:}\mathit{sig}_{\mathit{basis}}$ serving as an initial basis for the TSIL. The elaborator assumes Γ declares $\mathit{basis}{:}\mathit{sig}_{\mathit{basis}}$, 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 $\mathrm{BV}(sdecs_0)$ with scope $sbnds : sdecs; S$ and variables $\mathrm{BV}(sdecs)$ with scope S. We write $\mathrm{BV}(L)$ for $\mathrm{BV}(sdecs_0) \cup \mathrm{BV}(sdecs)$.

- 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 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 $sigid{=}sig, S$ to match the first binding in the list.
  - We usually omit the initial $\cdot$; for example,

$$sigid_1{=}sig_1, \ldots, sigid_1{=}sig_1.$$

- We define the domain of a signature abbreviation, $\mathrm{dom}(S)$, by

$$\begin{array}{lcl}
\mathrm{dom}(\cdot) & = & \emptyset \\
\mathrm{dom}(S, sigid{=}sig) & = & \mathrm{dom}(S) \cup \{sigid\} \\
\mathrm{dom}(S, unitid{=}S_u) & = & \mathrm{dom}(S) \cup \{unitid\}.
\end{array}$$

- We define the function $S{+\!\!+}S'$ by

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

  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.

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

$$\frac{\begin{array}{c} decs \vdash sdecs_0\ \mathsf{ok} \\ decs, sdecs_0 \vdash sbnds : sdecs \\ decs, sdecs_0, sdecs \vdash S\ \mathsf{ok} \\ sdecs_0 = lab_1 \triangleright var_1{:}[sdecs_1], \ldots, lab_n \triangleright var_n{:}[sdecs_n] \end{array}}{decs \vdash sdecs_0 \to sbnds : sdecs; S\ \mathsf{ok}} \tag{1}$$

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

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

$$\frac{\vdash decs\ \mathsf{ok}}{decs \vdash \cdot\ \mathsf{ok}} \tag{2}$$

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

$$\frac{\begin{array}{c} decs \vdash S'\ \mathsf{ok} \quad decs \vdash S\ \mathsf{ok} \quad unitid \notin \mathrm{dom}(S) \\ S' = (sigid_1{=}sig_1, \ldots, sigid_n{=}sig_n) \end{array}}{decs \vdash unitid{=}S', S\ \mathsf{ok}} \tag{4}$$

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

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

$$\frac{\begin{array}{c} L_{basis} := \cdot \to \mathsf{basis}{=}mod_{basis} : \mathsf{basis}{:}sig_{basis}; \cdot \\ \vdash L_{basis} {+\!\!+} L \rightsquigarrow L' \quad L' \rightsquigarrow exp : \{\} \end{array}}{L \rightsquigarrow exp : \{\}} \tag{6}$$

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

$$\frac{L = sdecs_0 \to sbnds : sdecs; S}{\begin{array}{l} decs \vdash L {+\!\!+} (\cdot \to sbnds' : sdecs'; S') \rightsquigarrow \\ \quad sdecs_0 \to sbnds {+\!\!+} sbnds' : sdecs {+\!\!+} sdecs'; S {+\!\!+} S' \end{array}} \tag{7}$$

$$\frac{\begin{array}{c} sdecs = sdecs'', lab \triangleright var'{:}sig', sdecs''' \\ decs, sdecs_0, sdecs \vdash_{\mathsf{sub}} var'{:}sig' \preceq sig \rightsquigarrow mod{:}sig'' \\ sbnd := 1 \triangleright var{=}mod \quad sdec := 1 \triangleright var{:}sig'' \\ L := sdecs_0 \to sbnds {+\!\!+} sbnd : sdecs {+\!\!+} sdec; S \\ decs \vdash L {+\!\!+} (sdecs_1 \to sbnds' : sdecs'; S') \rightsquigarrow L'' \end{array}}{\begin{array}{l} decs \vdash (sdecs_0 \to sbnds : sdecs; S) {+\!\!+} \\ \quad (lab \triangleright var{:}sig, sdecs_1 \to sbnds' : sdecs'; S') \rightsquigarrow L'' \end{array}} \tag{8}$$

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

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

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 $\mathrm{BV}(sdecs)$ with scope S. We define $\mathrm{BV}(\Gamma)$ by $\mathrm{BV}(sdecs)$. Second, we extend the syntax for TSEL signature expressions:

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

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

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

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.

$\boxed{project \rightsquigarrow L}$

$$\frac{}{\cdot \rightsquigarrow (\cdot \to \cdot : \cdot; \cdot)} \tag{11}$$

$$\frac{\begin{array}{c} \mathit{project} \rightsquigarrow L \quad \mathit{basis} \notin \mathrm{BV}(L) \\ L = \mathit{sdecs}_0 \rightarrow \mathit{sbnds} : \mathit{sdecs}; S \\ \Gamma := \mathit{basis}{:}\mathit{sig}_{\mathit{basis}}, R(\mathit{sdecs}_0), \mathit{sdecs}; S \\ \Gamma \vdash \mathit{srcunit} \rightsquigarrow L' \quad \mathit{var} \notin \mathrm{BV}(L') \\ \mathit{sdecs}_1' \rightarrow \mathit{sbnds}' : \mathit{sdecs}'; S' := \{\mathit{var}/\mathit{basis}\}L' \\ \mathit{sdecs}_1 := \mathsf{basis} \triangleright \mathit{var}{:}\mathit{sig}_{\mathit{basis}}, \mathit{sdecs}_1' \\ \vdash L +\!\!+ (\mathit{sdecs}_1 \rightarrow \mathit{sbnds}' : \mathit{sdecs}'; S') \rightsquigarrow L'' \end{array}}{\mathit{project}, \mathit{srcunit} \rightsquigarrow L''} \tag{12}$$

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

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

$\boxed{\Gamma \vdash \mathit{srcunit} \rightsquigarrow L}$

$$\frac{\begin{array}{c} \Gamma \vdash \mathit{topdec} \rightsquigarrow L \\ L = \mathit{sdecs}_0 \rightarrow \mathit{sbnds} : \mathit{sdecs}; S \\ \mathit{var} \notin \mathrm{BV}(\Gamma) \cup \mathrm{BV}(L) \\ \mathit{sbnds}' := \overline{\mathit{unitid}} \triangleright \mathit{var}{=}[\mathit{sbnds}] \\ \mathit{sdecs}' := \overline{\mathit{unitid}} \triangleright \mathit{var}{:}[\mathit{sdecs}] \\ S' := \mathit{unitid}{=}\sigma(\mathit{var}, \mathit{sdecs}, S) \end{array}}{\begin{array}{l} \Gamma \vdash \mathtt{unit}\ \mathit{unitid} = \mathit{topdec} \rightsquigarrow \\ \quad \mathit{sdecs}_0 \rightarrow \mathit{sbnds}' : \mathit{sdecs}'; S' \end{array}} \tag{14}$$

$\boxed{\Gamma \vdash \mathit{topdec} \rightsquigarrow L}$

$$\frac{\Gamma \vdash \mathit{impexp} \rightsquigarrow L}{\Gamma \vdash \mathtt{import}\ \mathit{impexp} \rightsquigarrow L} \tag{15}$$

$$\frac{\Gamma \vdash \mathit{strdec} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs}}{\Gamma \vdash \mathit{strdec} \rightsquigarrow \cdot \rightarrow \mathit{sbnds} : \mathit{sdecs}; \cdot} \tag{16}$$

$$\frac{\Gamma \vdash \mathit{sigbind} \rightsquigarrow S}{\Gamma \vdash \mathtt{signature}\ \mathit{sigbind} \rightsquigarrow \cdot \rightarrow \cdot : \cdot; S} \tag{17}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{topdec} \leadsto \mathit{sdecs}_0 \to \mathit{sbnds} : \mathit{sdecs}; S \\ \mathit{var} \notin \mathrm{BV}(\Gamma) \cup \mathrm{BV}(\mathit{sdecs}_0) \\ \Gamma, R(\mathit{sdecs}_0), 1^\star \rhd \mathit{var}{:}[\mathit{sdecs}], S \vdash \mathit{topdec}' \leadsto L' \\ L := \mathit{sdecs}_0 \to 1 \rhd \mathit{var}{=}[\mathit{sbnds}] : 1 \rhd \mathit{var}{:}[\mathit{sdecs}]; \cdot \\ \Gamma \vdash L {+\!\!+} L' \leadsto L''\end{array}}{\Gamma \vdash \texttt{local}\ \mathit{topdec}\ \texttt{in}\ \mathit{topdec}'\ \texttt{end} \leadsto L''} \tag{18}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{topdec} \leadsto L \\ L = \mathit{sdecs}_0 \to \mathit{sbnds} : \mathit{sdecs}; S \\ \Gamma, R(\mathit{sdecs}_0), \mathit{sdecs}, S \vdash \mathit{topdec}' \leadsto L' \\ \Gamma \vdash L {+\!\!+} L' \leadsto L''\end{array}}{\Gamma \vdash \mathit{topdec}\ \mathit{topdec}' \leadsto L''} \tag{19}$$

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

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

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

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

Rule 21: The signature *sig* should be fully selfified.

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

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

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

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

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

$$\frac{}{\mathit{sdecs}; S, \mathit{sigid}{=}\mathit{sig} \vdash \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{24}$$

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

$$\frac{\mathit{sdecs}; S \vdash \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}{\mathit{sdecs}; S, \mathit{unitid}{=}S' \vdash \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{26}$$

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

$$\frac{}{\mathit{sdecs}; S, \mathit{unitid}{=}S' \vdash \mathit{unitid} \rightsquigarrow S'} \tag{27}$$

$$\frac{\begin{array}{c}\mathit{unitid}' \neq \mathit{unitid} \\ \mathit{sdecs}; S \vdash \mathit{unitid} \rightsquigarrow S''\end{array}}{\mathit{sdecs}; S, \mathit{unitid}'{=}S' \vdash \mathit{unitid} \rightsquigarrow S''} \tag{28}$$

$$\frac{\mathit{sdecs}; S \vdash \mathit{unitid} \rightsquigarrow S'}{\mathit{sdecs}; S, \mathit{sigid}{=}\mathit{sig} \vdash \mathit{unitid} \rightsquigarrow S'} \tag{29}$$

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

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

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

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

$$
\begin{array}{rcl}
B \text{ or } T, F, G, E & \in & \mathrm{Basis} = \mathrm{TyNameSet} \times \mathrm{FunEnv} \times \mathrm{SigEnv} \times \mathrm{Env} \\
T & \in & \mathrm{TyNameSet} = \mathrm{Fin}(\mathrm{TyName}) \\
F & \in & \mathrm{FunEnv} = \mathrm{FunId} \stackrel{\mathrm{fin}}{\to} \mathrm{FunSig} \\
G & \in & \mathrm{SigEnv} = \mathrm{SigId} \stackrel{\mathrm{fin}}{\to} \mathrm{Sig} \\
E \text{ or } (SE, TE, VE) & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
\Phi \text{ or } (T)(E, (T')E') & \in & \mathrm{FunSig} = \mathrm{TyNameSet} \times (\mathrm{Env} \times \mathrm{Sig}) \\
\Sigma \text{ or } (T)E & \in & \mathrm{Sig} = \mathrm{TyNameSet} \times \mathrm{Env} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \stackrel{\mathrm{fin}}{\to} \mathrm{Env} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \stackrel{\mathrm{fin}}{\to} \mathrm{TyStr} \\
VE & \in & \mathrm{ValEnv} = \mathrm{VId} \stackrel{\mathrm{fin}}{\to} \mathrm{TypeScheme} \times \mathrm{IdStatus} \\
t & \in & \mathrm{TyName} \quad \text{(type names)} \\
\mathit{funid} & \in & \mathrm{FunId} \quad \text{(functor identifiers)} \\
\mathit{sigid} & \in & \mathrm{SigId} \quad \text{(signature identifiers)} \\
\mathit{strid} & \in & \mathrm{StrId} \quad \text{(structure identifiers)} \\
\mathit{tycon} & \in & \mathrm{TyCon} \quad \text{(type constructors)} \\
\mathit{vid} & \in & \mathrm{VId} \quad \text{(value identifiers)}
\end{array}
$$

Figure 20: Static TDIL for Standard ML (summary). Fin(A) 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 wf for "every type structure occuring in A is well-formed" [16, Section 4.9].

  We assume familiarity with realizations φ and their support Supp φ [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 $\{a \in A \;;\; a \notin B\}$.

  We write $\#A$ for the cardinality of the finite set A.

- **Finite maps.** The domain of a finite map $f : A \stackrel{\mathrm{fin}}{\to} B$ is the set 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 $\{a \mapsto b \;;\; \phi\}$ for the map taking every a satisfying ϕ to $b(a)$.

- **Extension.** To extend finite maps $f, g : A \stackrel{\mathrm{fin}}{\to} B$, we define $f + g : A \stackrel{\mathrm{fin}}{\to} B$ by

$$
(f+g)(a) = \begin{cases} g(a) & \text{if } a \in \mathrm{dom}(g) \\ f(a) & \text{otherwise.} \end{cases}
$$

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,

$$\begin{aligned}
&\Gamma + U = B, U' {+} U \\
&\Gamma + B' = B {+} B', U' \\
&\Gamma + T = B {+} T, U' = (T' {+} T, F, G, E), U'
\end{aligned}$$

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,

$$\begin{aligned}
&\Gamma(\mathit{unitid}) = U(\mathit{unitid}) \\
&B(\mathit{strid}) = E(\mathit{strid}) = \mathit{SE}(\mathit{strid})
\end{aligned}$$

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

- **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 $B = T', F, G, E$.

$\boxed{\Gamma \vdash \mathit{udecs} : \mathit{uspecs}}$

$$\frac{\Gamma \ \mathsf{ok}}{\Gamma \vdash \cdot : \cdot} \tag{33}$$

$$\frac{\begin{array}{c}\mathit{srcunit} = (\texttt{unit}\ \mathit{unitid} = \texttt{unit}\ \mathit{topdec}\ \texttt{end}) \quad \Upsilon = (T)(F, G, E) \\ \Gamma \vdash \mathit{topdec} : T, F, G, E \\ \Gamma + T + \{\mathit{unitid} \mapsto T, F, G, E\} \vdash \mathit{udecs} : \mathit{uspecs}\end{array}}{\Gamma \vdash (\mathit{srcunit}, \mathit{udecs}) : (\mathit{unitid}{:}\Upsilon, \mathit{uspecs})} \tag{34}$$

$\boxed{\Gamma \vdash \mathit{topdec} : B}$

$$\frac{\Gamma \vdash \mathit{impexp} : F, G, E}{\Gamma \vdash \texttt{import}\ \mathit{impexp} : \emptyset, F, G, E} \tag{35}$$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \quad T := \text{tynames } E \setminus (T \text{ of } B) \quad \text{tyvars } E = \emptyset}{B, U \vdash \mathit{strdec} : T, \{\}, \{\}, E} \tag{36}$$

$$\frac{B \vdash \mathit{sigdec} \Rightarrow G}{B, U \vdash \mathit{sigdec} : \emptyset, \{\}, G, E_\emptyset} \tag{37}$$

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

$$\frac{B \vdash \mathit{fundec} \Rightarrow F \quad T := \text{tynames } F \setminus (T \text{ of } B) \quad \text{tyvars } F = \emptyset}{B, U \vdash \mathit{fundec} : T, F, \{\}, E_\emptyset} \tag{38}$$

$$\frac{\Gamma \vdash \mathit{topdec} : B \quad \Gamma + B \vdash \mathit{topdec}' : T', F', G', E' \quad T := (T \text{ of } B) \cup T'}{\Gamma \vdash \texttt{local}\ \mathit{topdec}\ \texttt{in}\ \mathit{topdec}'\ \texttt{end} : T, F', G', E'} \tag{39}$$

$$\frac{\Gamma \vdash \mathit{topdec} : B \quad \Gamma + B \vdash \mathit{topdec}' : B'}{\Gamma \vdash \mathit{topdec}\ \mathit{topdec}' : B + B'} \tag{40}$$

$\boxed{\Gamma \vdash \mathit{impexp} : F, G, E}$

$$\frac{\Gamma(\mathit{unitid}) = T, F, G, E}{\Gamma \vdash \mathit{unitid} : F, G, E} \tag{41}$$

$$\frac{\begin{array}{c} B \text{ of } \Gamma \vdash \mathit{intexp} \Rightarrow \Upsilon \quad \Upsilon = (T)(F, \{\}, E) \\ \Gamma(\mathit{unitid}) = T', F', G', E' \quad F', G', E' : \Upsilon \ \mathsf{using}\ \varphi \end{array}}{\Gamma \vdash (\mathit{unitid} : \mathit{intexp}) : \varphi(F), \{\}, \varphi(E)} \tag{42}$$

$$\frac{\Gamma \vdash \mathit{impexp} : F, G, E \quad \Gamma \vdash \mathit{impexp}' : F', G', E'}{\Gamma \vdash \mathit{impexp}\ \mathit{impexp}' : F{+}F', G{+}G', E{+}E'} \tag{43}$$

$\boxed{B \vdash \mathit{intexp} \Rightarrow \Upsilon}$

$$\frac{B \vdash \mathit{topspec} \Rightarrow F, E \quad T := (\text{tynames } F, E) \setminus (T \text{ of } B)}{B \vdash \texttt{intf}\ \mathit{topspec}\ \texttt{end} \Rightarrow (T)(F, \{\}, E)} \tag{44}$$

$\boxed{B \vdash \mathit{topspec} \Rightarrow F, E}$

$$\frac{B \vdash \mathit{spec} \Rightarrow E}{B \vdash \mathit{spec} \Rightarrow \{\}, E} \tag{45}$$

$$\frac{B \vdash \mathit{funspec} \Rightarrow F}{B \vdash \texttt{functor}\ \mathit{funspec} \Rightarrow F, \{\}} \tag{46}$$

$$\frac{\begin{array}{c} B \vdash \mathit{topspec} \Rightarrow F, E \quad B + E \vdash \mathit{topspec}' \Rightarrow F', E' \\ \text{dom}(F) \cap \text{dom}(F') = \emptyset \quad \text{dom}(E) \cap \text{dom}(E') = \emptyset \end{array}}{B \vdash \mathit{topspec}\ \mathit{topspec}' \Rightarrow F{+}F', E{+}E'} \tag{47}$$

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

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow (T)E \quad B + T + \{\mathit{strid} \mapsto E\} \vdash \mathit{sigexp}' \Rightarrow (T')E' \\ F := \{\mathit{funid} \mapsto (T)(E, (T')E')\} \\ \langle B \vdash \mathit{funspec} \Rightarrow F' \quad \mathit{funid} \notin \mathrm{dom}(F') \rangle \end{array}}{B \vdash \mathit{funid}(\mathit{strid} : \mathit{sigexp}) : \mathit{sigexp}' \ \langle \texttt{and}\ \mathit{funspec} \rangle \Rightarrow F \langle + F' \rangle} \tag{48}$$

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

$$\frac{\text{tynames } F, G, E, U \subset T \quad F, G, E, U \ \mathsf{wf}}{(T, F, G, E), U \ \mathsf{ok}} \tag{49}$$

$\boxed{\Gamma \vdash \Upsilon : \mathsf{Sig}}$

$$\frac{\begin{array}{c} T = T \text{ of } \Gamma \\ T \cap T' = \emptyset \quad \text{tynames } F, G, E \subset T \cup T' \\ \text{tyvars } F, G, E = \emptyset \quad F, G, E \ \mathsf{wf} \end{array}}{\Gamma \vdash (T')(F, G, E) : \mathsf{Sig}} \tag{50}$$

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

$$\frac{\Gamma \ \mathsf{ok}}{\Gamma \vdash \cdot \ \mathsf{ok}} \tag{51}$$

$$\frac{\Gamma \vdash \Upsilon : \mathsf{Sig} \quad \Upsilon = (T')(F, G, E) \quad \Gamma + T' \vdash \mathit{uspecs} \ \mathsf{ok}}{\Gamma \vdash \mathit{unitid}{:}\Upsilon, \mathit{uspecs} \ \mathsf{ok}} \tag{52}$$

$\boxed{E : \Sigma \ \mathsf{using} \ \varphi}$

$$\frac{\Sigma \geq E^- \ \mathsf{using} \ \varphi \quad E \succ E^-}{E : \Sigma \ \mathsf{using} \ \varphi} \tag{53}$$

$\boxed{\Sigma \equiv \Sigma' \ \mathsf{using} \ \varphi}$

$$\frac{T = \varphi(T') \quad E = \varphi(E') \quad \text{Supp } \varphi \subset T' \quad \#T = \#T'}{(T)E \equiv (T')E' \ \mathsf{using} \ \varphi} \tag{54}$$

$\boxed{\Upsilon \equiv \Upsilon' \text{ using } \varphi}$

$$\frac{(T)E \equiv (T')E' \text{ using } \varphi \quad F = \varphi(F') \quad G = \varphi(G')}{(T)(F, G, E) \equiv (T')(F', G', E') \text{ using } \varphi} \tag{55}$$

$\boxed{F, G, E : \Upsilon \text{ using } \varphi}$

$$\frac{\begin{array}{l} E : (T)E' \text{ using } \varphi \\ \mathrm{dom}(F) \supset \mathrm{dom}(F') \\ \forall \mathit{funid} \in \mathrm{dom}(F').\, \exists \varphi_1, \varphi_2 \left\{ \begin{array}{l} E'_1 : (T_1)E_1 \text{ using } \varphi_1 \\ \varphi_1(E_2) : (T'_2)E'_2 \text{ using } \varphi_2 \\ \text{where } (T'_1)(E'_1, (T'_2)E'_2) = \varphi(F'(\mathit{funid})) \\ \text{and } (T_1)(E_1, (T_2)E_2) = F(\mathit{funid}) \end{array} \right. \\ \mathrm{dom}(G) \supset \mathrm{dom}(G') \\ \forall \mathit{sigid} \in \mathrm{dom}(G').\, \exists \varphi'.\, G(\mathit{sigid}) \equiv \varphi(G'(\mathit{sigid})) \text{ using } \varphi' \end{array}}{F, G, E : (T)(F', G', E') \text{ using } \varphi} \tag{56}$$

$$
\begin{array}{rcl}
(F, G, E) \text{ or } B & \in & \text{Basis} = \text{FunEnv} \times \text{SigEnv} \times \text{Env} \\
F & \in & \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunctorClosure} \\
G & \in & \text{SigEnv} = \text{SigId} \xrightarrow{\text{fin}} \text{Int} \\
(\mathit{SE}, \mathit{TE}, \mathit{VE}) \text{ or } E & \in & \text{Env} = \text{StrEnv} \times \text{TyEnv} \times \text{ValEnv} \\
(\mathit{strid} : I, \mathit{strexp}, B) & \in & \text{FunctorClosure} = (\text{StrId} \times \text{Int}) \times \text{StrExp} \times \text{Basis} \\
(\mathit{SI}, \mathit{TI}, \mathit{VI}) \text{ or } I & \in & \text{Int} = \text{StrInt} \times \text{TyInt} \times \text{ValInt} \\
\mathit{SE} & \in & \text{StrEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{Env} \\
\mathit{TE} & \in & \text{TyEnv} = \text{TyCon} \xrightarrow{\text{fin}} \text{ValEnv} \\
\mathit{VE} & \in & \text{ValEnv} = \text{VId} \xrightarrow{\text{fin}} \text{Val} \times \text{IdStatus} \\
\mathit{SI} & \in & \text{StrInt} = \text{StrId} \xrightarrow{\text{fin}} \text{Int} \\
\mathit{TI} & \in & \text{TyInt} = \text{TyCon} \xrightarrow{\text{fin}} \text{ValInt} \\
\mathit{VI} & \in & \text{ValInt} = \text{VId} \xrightarrow{\text{fin}} \text{IdStatus} \\
(G, I) \text{ or } \mathit{IB} & \in & \text{IntBasis} = \text{SigEnv} \times \text{Int}
\end{array}
$$

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 FunctorClosure as follows, permitting an optional ascribed interface on the functor body.

  $$
  \begin{array}{rcl}
  & & \text{FunctorClosure} = \text{FClos} \cup \text{FClos}' \\
  (\mathit{strid} : I, \mathit{strexp}, B) & \in & \text{FClos} = (\text{StrId} \times \text{Int}) \times \text{StrExp} \times \text{Basis} \\
  (\mathit{strid} : I, \mathit{strexp} : I', B) & \in & \text{FClos}' = (\text{StrId} \times \text{Int}) \times (\text{StrExp} \times \text{Int}) \times \text{Basis.}
  \end{array}
  $$

  Here, $A \cup B$ denotes the disjoint union of A and B.
- We define the function $\downarrow$: Basis $\times$ UnitInt $\rightarrow$ Basis that cuts down a basis B to match a unit interface UI:

  $$
  \begin{array}{l}
  \downarrow : \text{Basis} \times \text{UnitInt} \rightarrow \text{Basis} \\
  (F, G, E) \downarrow (\mathit{FI}, I) = (F \downarrow \mathit{FI}, \{\}, E \downarrow I) \\
  \\
  \downarrow : \text{FunEnv} \times \text{FunIntEnv} \rightarrow \text{FunEnv} \\
  F \downarrow \mathit{FI} = \{\mathit{funid} \mapsto F(\mathit{funid}) \downarrow \mathit{FI}(\mathit{funid}) \ ; \ \mathit{funid} \in \text{dom}(F) \cap \text{dom}(\mathit{FI})\} \\
  \\
  \downarrow : \text{FunctorClosure} \times (\text{Int} \times \text{Int}) \rightarrow \text{FunctorClosure} \\
  (\mathit{strid}{:}I_0, \mathit{strexp}, B) \downarrow (I, I') = (\mathit{strid}{:}I, \mathit{strexp}{:}I', B) \\
  (\mathit{strid}{:}I_0, \mathit{strexp}{:}I_0', B) \downarrow (I, I') = (\mathit{strid}{:}I, \mathit{strexp}{:}I', B).
  \end{array}
  $$
- TD's evaluation judgement $s, B \vdash \mathit{strexp} \Rightarrow E/p, s'$ handles functor application for functor closures of the form FClos. We extend that judgement by adding the following rules to handle

functor closures of the form FClos$'$.

$$\frac{\begin{array}{c} B(\mathit{funid}) = (\mathit{strid}{:}I, \mathit{strexp}'{:}I', B') \\ s, B \vdash \mathit{strexp} \Rightarrow E, s' \quad s', B' + \{\mathit{strid} \mapsto E \downarrow I\} \vdash \mathit{strexp}' \Rightarrow E', s'' \end{array}}{s, B \vdash \mathit{funid}(\mathit{strexp}) \Rightarrow E' \downarrow I', s''}$$

$$\frac{\begin{array}{c} B(\mathit{funid}) = (\mathit{strid}{:}I, \mathit{strexp}'{:}I', B') \\ s, B \vdash \mathit{strexp} \Rightarrow p, s' \end{array}}{s, B \vdash \mathit{funid}(\mathit{strexp}) \Rightarrow p, s'}$$

$$\frac{\begin{array}{c} B(\mathit{funid}) = (\mathit{strid}{:}I, \mathit{strexp}'{:}I', B') \\ s, B \vdash \mathit{strexp} \Rightarrow E, s' \quad s', B' + \{\mathit{strid} \mapsto E \downarrow I\} \vdash \mathit{strexp}' \Rightarrow p, s'' \end{array}}{s, B \vdash \mathit{funid}(\mathit{strexp}) \Rightarrow p, s''}$$

$\boxed{s, \Gamma \vdash \mathit{udecs} \Rightarrow \Gamma'/p, s'}$

$$\frac{}{s, \Gamma \vdash \cdot \Rightarrow \Gamma, s} \tag{57}$$

$$\frac{\begin{array}{c} \mathit{srcunit} = (\texttt{unit}\ \mathit{unitid} = \texttt{unit}\ \mathit{topdec}\ \texttt{end}) \\ s, \Gamma \vdash \mathit{topdec} \Rightarrow B, s' \quad s', \Gamma + \{\mathit{unitid} \mapsto B\} \vdash \mathit{udecs} \Rightarrow \Gamma', s'' \end{array}}{s, \Gamma \vdash \mathit{srcunit}, \mathit{udecs} \Rightarrow \Gamma', s''} \tag{58}$$

$$\frac{\begin{array}{c} \mathit{srcunit} = (\texttt{unit}\ \mathit{unitid} = \texttt{unit}\ \mathit{topdec}\ \texttt{end}) \\ s, \Gamma \vdash \mathit{topdec} \Rightarrow p, s' \end{array}}{s, \Gamma \vdash \mathit{srcunit}, \mathit{udecs} \Rightarrow p, s'} \tag{59}$$

$$\frac{\begin{array}{c} \mathit{srcunit} = (\texttt{unit}\ \mathit{unitid} = \texttt{unit}\ \mathit{topdec}\ \texttt{end}) \\ s, \Gamma \vdash \mathit{topdec} \Rightarrow B, s' \quad s', \Gamma + \{\mathit{unitid} \mapsto B\} \vdash \mathit{udecs} \Rightarrow p, s'' \end{array}}{s, \Gamma \vdash \mathit{srcunit}, \mathit{udecs} \Rightarrow p, s''} \tag{60}$$

$\boxed{s, \Gamma \vdash \mathit{topdec} \Rightarrow B/p, s'}$

$$\frac{\Gamma \vdash \mathit{impexp} \Rightarrow B}{s, \Gamma \vdash \texttt{import}\ \mathit{impexp} \Rightarrow B, s} \tag{61}$$

$$\frac{s, B \vdash \mathit{strdec} \Rightarrow E, s'}{s, (B, U) \vdash \mathit{strdec} \Rightarrow (\{\}, \{\}, E), s'} \tag{62}$$

$$\frac{s, B \vdash \mathit{strdec} \Rightarrow p, s'}{s, (B, U) \vdash \mathit{strdec} \Rightarrow p, s'} \tag{63}$$

$$\frac{\mathrm{Inter}\ B \vdash \mathit{sigdec} \Rightarrow G}{s, (B, U) \vdash \mathit{sigdec} \Rightarrow (\{\}, G, E_\emptyset), s} \tag{64}$$

$$\frac{B \vdash \mathit{fundec} \Rightarrow F}{s, (B, U) \vdash \mathit{fundec} \Rightarrow (F, \{\}, E_\emptyset), s} \tag{65}$$

$$\frac{s, \Gamma \vdash \mathit{topdec} \Rightarrow B, s' \quad s', \Gamma{+}B \vdash \mathit{topdec}' \Rightarrow B', s''}{s, \Gamma \vdash \texttt{local}\ \mathit{topdec}\ \texttt{in}\ \mathit{topdec}'\ \texttt{end} \Rightarrow B', s''} \tag{66}$$

$$\frac{s, \Gamma \vdash \mathit{topdec} \Rightarrow p, s'}{s, \Gamma \vdash \texttt{local}\ \mathit{topdec}\ \texttt{in}\ \mathit{topdec}'\ \texttt{end} \Rightarrow p, s'} \tag{67}$$

$$\frac{s, \Gamma \vdash \mathit{topdec} \Rightarrow B, s' \quad s', \Gamma{+}B \vdash \mathit{topdec}' \Rightarrow p, s''}{s, \Gamma \vdash \texttt{local}\ \mathit{topdec}\ \texttt{in}\ \mathit{topdec}'\ \texttt{end} \Rightarrow p, s''} \tag{68}$$

$$\frac{s, \Gamma \vdash \mathit{topdec} \Rightarrow B, s' \quad s', \Gamma{+}B \vdash \mathit{topdec}' \Rightarrow B', s''}{s, \Gamma \vdash \mathit{topdec}\ \ \mathit{topdec}' \Rightarrow B{+}B', s''} \tag{69}$$

$$\frac{s, \Gamma \vdash \mathit{topdec} \Rightarrow p, s'}{s, \Gamma \vdash \mathit{topdec}\ \ \mathit{topdec}' \Rightarrow p, s'} \tag{70}$$

$$\frac{s, \Gamma \vdash \mathit{topdec} \Rightarrow B, s' \quad s', \Gamma{+}B \vdash \mathit{topdec}' \Rightarrow p, s''}{s, \Gamma \vdash \mathit{topdec}\ \ \mathit{topdec}' \Rightarrow p, s''} \tag{71}$$

$\boxed{\Gamma \vdash \mathit{impexp} \Rightarrow B}$

$$\frac{\Gamma(\mathit{unitid}) = B'}{\Gamma \vdash \mathit{unitid} \Rightarrow B'} \tag{72}$$

$$\frac{\text{Inter } B \vdash \mathit{topspec} \Rightarrow \mathit{UI} \quad U(\mathit{unitid}) = B'}{B, U \vdash \mathit{unitid} : \texttt{intf}\ \mathit{topspec}\ \texttt{end} \Rightarrow B' \downarrow \mathit{UI}} \tag{73}$$

$$\frac{\Gamma \vdash \mathit{impexp} \Rightarrow B \quad \Gamma \vdash \mathit{impexp}' \Rightarrow B'}{\Gamma \vdash \mathit{impexp}\ \ \mathit{impexp}' \Rightarrow B + B'} \tag{74}$$

$\boxed{\mathit{IB} \vdash \mathit{topspec} \Rightarrow \mathit{UI}}$

$$\frac{\mathit{IB} \vdash \mathit{spec} \Rightarrow I}{\mathit{IB} \vdash \mathit{spec} \Rightarrow \{\}, I} \tag{75}$$

$$\frac{\mathit{IB} \vdash \mathit{funspec} \Rightarrow \mathit{FI}}{\mathit{IB} \vdash \texttt{functor}\ \mathit{funspec} \Rightarrow \mathit{FI}, (\{\}, \{\}, \{\})} \tag{76}$$

$$\frac{\mathit{IB} \vdash \mathit{topspec} \Rightarrow \mathit{FI}, I \quad \mathit{IB} + I \vdash \mathit{topspec}' \Rightarrow \mathit{FI}', I'}{\mathit{IB} \vdash \mathit{topspec}\ \ \mathit{topspec}' \Rightarrow \mathit{FI}{+}\mathit{FI}', I{+}I'} \tag{77}$$

$\boxed{IB \vdash \mathit{funspec} \Rightarrow \mathit{FI}}$

$$\frac{\begin{array}{c} IB \vdash \mathit{sigexp} \Rightarrow I \quad IB + \{\mathit{strid} \mapsto I\} \vdash \mathit{sigexp}' \Rightarrow I' \\ \mathit{FI} := \{\mathit{funid} \mapsto (I, I')\} \quad \langle IB \vdash \mathit{funspec} \Rightarrow \mathit{FI}' \rangle \end{array}}{IB \vdash \mathit{funid}(\mathit{strid} : \mathit{sigexp}) : \mathit{sigexp}' \ \langle \texttt{and}\ \mathit{funspec} \rangle \Rightarrow \mathit{FI} \langle + \mathit{FI}' \rangle} \tag{78}$$

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

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

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

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

$$\frac{\begin{array}{c} \Gamma \vdash \mathit{uspecs}\ \mathsf{ok} \\ \Gamma + \mathit{uspecs} \vdash \mathit{exps}\ \mathsf{ok} \\ \mathit{uspecs} = \mathit{unitid}_1{:}\Upsilon_1, \ldots, \mathit{unitid}_n{:}\Upsilon_n \\ \mathit{unitid}_1, \ldots, \mathit{unitid}_n \text{ are distinct} \end{array}}{\Gamma \vdash \mathit{uspecs} \to \mathit{exps}\ \mathsf{ok}} \tag{79}$$

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

$$\frac{\Gamma \vdash \mathit{udecs} : \mathit{uspecs}}{\Gamma \vdash \mathit{udecs} : \mathit{uspecs}\ \mathsf{ok}} \tag{80}$$

$$\frac{\Gamma \vdash (T)(F, G, E) : \mathsf{Sig}}{\Gamma \vdash T, F, G, E\ \mathsf{ok}} \tag{81}$$

$\boxed{L \Rightarrow \mathit{udecs}}$

$$\frac{}{\cdot \to \mathit{udecs} : \mathit{uspecs} \Rightarrow \mathit{udecs}} \tag{82}$$

$$\frac{\Upsilon \equiv \Upsilon_{\mathit{basis}}\ \mathsf{using}\ \varphi}{\mathsf{basis}{:}\Upsilon \to \mathit{udecs} : \mathit{uspecs} \Rightarrow \mathit{udecs}} \tag{83}$$

$\boxed{\Gamma \vdash L +\!\!+ L' \Rightarrow L''}$

$$\frac{\Gamma \vdash \mathit{exps} +\!\!+ \mathit{exps}' \Rightarrow \mathit{exps}''}{\Gamma \vdash (\mathit{uspecs}_0 \to \mathit{exps}) +\!\!+ (\cdot \to \mathit{exps}') \Rightarrow \mathit{uspecs}_0 \to \mathit{exps}''} \tag{84}$$

$$\frac{\begin{array}{c} \mathit{unitid} \in \mathrm{dom}(\mathit{uspecs}) \\ (\Gamma + \mathit{uspecs}_0 + \mathit{uspecs})(\mathit{unitid}) = T', F', G', E' \\ F', G', E' : \Upsilon \ \mathsf{using}\ \varphi \\ L' := \varphi(\mathit{uspecs}_1 \to \mathit{udecs}' : \mathit{uspecs}') \\ \Gamma \vdash (\mathit{uspecs}_0 \to \mathit{udecs} : \mathit{uspecs}) +\!\!+ L' \Rightarrow L'' \end{array}}{\Gamma \vdash (\mathit{uspecs}_0 \to \mathit{udecs} : \mathit{uspecs}) +\!\!+ (\mathit{unitid}{:}\Upsilon, \mathit{uspecs}_1 \to \mathit{udecs}' : \mathit{uspecs}') \Rightarrow L''} \tag{85}$$

$$\frac{\begin{array}{c} \mathit{unitid} \notin \mathrm{dom}(\mathit{exps}) \\ \mathit{uspecs}_0 = \mathit{uspecs}'', \mathit{unitid}{:}\Upsilon', \mathit{uspecs}''' \\ \Upsilon' \equiv \Upsilon \ \mathsf{using}\ \varphi \\ L' := \varphi(\mathit{uspecs}_1 \to \mathit{exps}') \\ \Gamma \vdash (\mathit{uspecs}_0 \to \mathit{exps}) +\!\!+ L' \Rightarrow L'' \end{array}}{\Gamma \vdash (\mathit{uspecs}_0 \to \mathit{exps}) +\!\!+ (\mathit{unitid}{:}\Upsilon, \mathit{uspecs}_1 \to \mathit{exps}') \Rightarrow L''} \tag{86}$$

$$\frac{\begin{array}{c} \mathit{unitid} \notin \mathrm{dom}(\mathit{exps}) \cup \mathrm{dom}(\mathit{uspecs}_0) \\ \Upsilon' \equiv \Upsilon \ \mathsf{using}\ \varphi \\ \Gamma + \mathrm{BT}(\mathit{uspecs}_0) \vdash \Upsilon' : \mathsf{Sig} \\ L := \mathit{uspecs}_0, \mathit{unitid}{:}\Upsilon' \to \mathit{exps} \\ L' := \varphi(\mathit{uspecs}_1 \to \mathit{exps}') \\ \Gamma \vdash L +\!\!+ L' \Rightarrow L'' \end{array}}{\Gamma \vdash (\mathit{uspecs}_0 \to \mathit{exps}) +\!\!+ (\mathit{unitid}{:}\Upsilon, \mathit{uspecs}_1 \to \mathit{exps}') \Rightarrow L''} \tag{87}$$

$\boxed{\Gamma \vdash \mathit{exps} +\!\!+ \mathit{exps}' \Rightarrow \mathit{exps}''}$

$$\frac{}{\Gamma \vdash (\mathit{udecs} : \mathit{uspecs}) +\!\!+ (\mathit{udecs}' : \mathit{uspecs}') \Rightarrow \mathit{udecs}, \mathit{udecs}' : \mathit{uspecs}, \mathit{uspecs}'} \tag{88}$$

$$\frac{}{\Gamma \vdash B +\!\!+ B' \Rightarrow B + B'} \tag{89}$$

F TD Elaboration Rules

We define the bound type names in a linkset, BT(L), by

$$\begin{array}{lcl} \mathrm{BT}(\mathit{uspecs}_0 \to \mathit{udecs} : \mathit{uspecs}) & = & \mathrm{BT}(\mathit{uspecs}_0) \cup \mathrm{BT}(\mathit{uspecs}) \\ \mathrm{BT}(\mathit{uspecs}_0 \to B) & = & \mathrm{BT}(\mathit{uspecs}_0). \end{array}$$

$\boxed{\mathit{project} \Rightarrow L}$

$$\frac{}{\cdot \Rightarrow \cdot \to \cdot} \tag{90}$$

$$\frac{\begin{array}{c} \mathit{project} \Rightarrow L \quad \Upsilon_{\mathit{basis}} = (T)(F,G,E) \quad T \cap \mathrm{BT}(L) = \emptyset \\ L = \mathit{uspecs}_0 \to \mathit{udecs} : \mathit{uspecs} \\ \Gamma := ((T,F,G,E),\{\}) + \mathrm{BT}(\mathit{uspecs}_0) + \mathit{uspecs} \\ \Gamma \vdash \mathit{srcunit} \Rightarrow \mathit{uspecs}_1 \to \mathit{udecs}' : \mathit{uspecs}' \\ \vdash L \mathbin{+\!\!+} (\mathsf{basis}{:}\Upsilon_{\mathit{basis}}, \mathit{uspecs}_1 \to \mathit{udecs}' : \mathit{uspecs}') \Rightarrow L'' \end{array}}{\mathit{project}, \mathit{srcunit} \Rightarrow L''} \tag{91}$$

$$\frac{\begin{array}{c} \mathit{project} \Rightarrow L \\ \mathrm{BT}(L) \cap \mathrm{BT}(L') = \emptyset \quad \vdash L' \ \mathsf{ok} \quad \vdash L \mathbin{+\!\!+} L' \Rightarrow L'' \end{array}}{\mathit{project}, L' \Rightarrow L''} \tag{92}$$

$\boxed{\Gamma \vdash \mathit{srcunit} \Rightarrow L}$

$$\frac{\begin{array}{c} \mathit{srcunit} = (\texttt{unit}\ \mathit{unitid} = \texttt{unit}\ \mathit{topdec}\ \texttt{end}) \\ \Gamma \vdash \mathit{topdec} \Rightarrow \mathit{uspecs}_0 \to T,F,G,E \end{array}}{\Gamma \vdash \mathit{srcunit} \Rightarrow \mathit{uspecs}_0 \to \mathit{srcunit} : \mathit{unitid}{:}(T)(F,G,E)} \tag{93}$$

$\boxed{\Gamma \vdash \mathit{topdec} \Rightarrow L}$

$$\frac{\Gamma \vdash \mathit{impexp} \Rightarrow L}{\Gamma \vdash \texttt{import}\ \mathit{impexp} \Rightarrow L} \tag{94}$$

$$\frac{\mathit{topdec} = \mathit{strdec} \quad \Gamma \vdash \mathit{topdec} : B}{\Gamma \vdash \mathit{topdec} \Rightarrow \cdot \to B} \tag{95}$$

$$\frac{\mathit{topdec} = \mathit{sigdec} \quad \Gamma \vdash \mathit{topdec} : B}{\Gamma \vdash \mathit{topdec} \Rightarrow \cdot \to B} \tag{96}$$

$$\frac{\mathit{topdec} = \mathit{fundec} \quad \Gamma \vdash \mathit{fundec} : B}{\Gamma \vdash \mathit{topdec} \Rightarrow \cdot \to B} \tag{97}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathit{topdec} \Rightarrow \mathit{uspecs}_0 \to B \\ \Gamma + \mathrm{BT}(\mathit{uspecs}_0) + B \vdash \mathit{topdec}' \Rightarrow L \\ B' := T \text{ of } B, \{\}, \{\}, E_\emptyset \quad \Gamma \vdash (\mathit{uspecs}_0 \to B') \mathbin{+\!\!+} L \Rightarrow L' \end{array}}{\Gamma \vdash \texttt{local}\ \mathit{topdec}\ \texttt{in}\ \mathit{topdec}'\ \texttt{end} \Rightarrow L'} \tag{98}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathit{topdec} \Rightarrow L \quad L = \mathit{uspecs}_0 \to B \\ \Gamma + \mathrm{BT}(\mathit{uspecs}_0) + B \vdash \mathit{topdec}' \Rightarrow L' \\ \Gamma \vdash L \mathbin{+\!\!+} L' \Rightarrow L'' \end{array}}{\Gamma \vdash \mathit{topdec}\ \mathit{topdec}' \Rightarrow L''} \tag{99}$$

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

$$\frac{\Gamma(\mathit{unitid}) = T, F, G, E \quad L := \mathit{unitid}{:}(T)(F, G, E) \to \emptyset, F, G, E}{\Gamma \vdash \mathit{unitid} \Rightarrow L} \tag{100}$$

$$\frac{\Gamma \vdash \mathit{intexp} \Rightarrow (T)(F, G, E) \quad L := \mathit{unitid}{:}(T)(F, G, E) \to \emptyset, F, G, E}{\Gamma \vdash \mathit{unitid} : \mathit{intexp} \Rightarrow L} \tag{101}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{impexp} \Rightarrow L \quad \Gamma \vdash \mathit{impexp}' \Rightarrow L' \\ \mathrm{BT}(L) \cap \mathrm{BT}(L') = \emptyset \quad \Gamma \vdash L {+\!\!+} L' \Rightarrow L''\end{array}}{\Gamma \vdash \mathit{impexp}\ \mathit{impexp}' \Rightarrow L''} \tag{102}$$

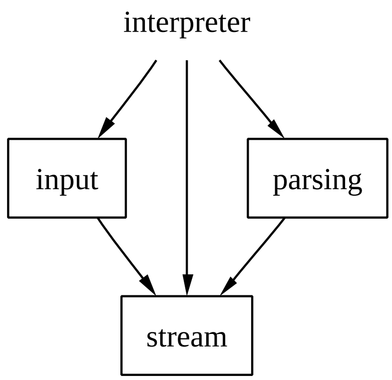

Figure 22: The dependencies among the libraries and the Mini-ML interpreter. The interpreter uses all three libraries and the input and parsing libraries use the stream library.

G Larger Example: Mini-ML

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 `BASIC_STREAM` describing the visible "core" of functional streams, a signature `STREAM` that extends `BASIC_STREAM`, a functor `Stream` that lifts a `BASIC_STREAM` structure to `STREAM`, and plain and memoizing `STREAM` structures. The handoff unit `StreamLib` declares all of these components by importing units `STREAMSIG` and `StreamImpl`:

```
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
```

[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 :
    intf
      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
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`:

```
unit INPUTSIG =
unit
  local
    import StreamLib
  in
    signature INPUT =
      sig
        val readfile  : string -> char Stream.stream
        val readkeybd : unit   -> char Stream.stream
        val promptkeybd : string -> char Stream.stream
      end
  end
end

unit InputLib =
unit
  import INPUTSIG

  import InputImpl :
    intf
      structure Input : INPUT
    end
end
```

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,

```
Pos.markstream : char Stream.stream -> (char * Pos.pos) Stream.stream
```

marks a character stream with position information for reporting errors and

```
Parsing.transform : ('a,'t) Parsing.parser
  -> ('t * Pos.pos) Stream.stream
  -> '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
```

```
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
```

Interperter

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
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`:

```
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`.

```
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 auxilliary 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

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 `Interp` and `RunInterp`:

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

We then compile the libraries:

$$\begin{aligned} L_S &= link(STREAMSIG, StreamImpl)\\ L_I &= link(STREAMSIG, StreamLib, INPUTSIG, TextIOLib, InputImpl)\\ L_P &= link(STREAMSIG, StreamLib,\\ &\qquad PARSINGSIG, PosImpl, ParsingFunImpl, ParsingImpl) \end{aligned}$$

Finally, we complete the linkset

$$L = link(L_S, L_I, L_P, L_r)$$

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.

```
> 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
>
```

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

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

to avoid recompiling `STREAMSIG` and `StreamLib` while producing $L_S = L_1$, $L_I = L_3$, and $L_P = L_4$.