Separate Compilation for Standard ML

David Swasey, Tom Murphy VII, Karl Crary, Robert Harper
Carnegie Mellon University

October 30, 2005

Abstract

This is a proposal for an extension to the Standard ML programming language for programming "in the large." The extension allows the programmer to write a program broken into multiple fragments in way that would be compatible between different implementations. It also allows for the separate compilation of these fragments and for incremental recompilation strategies such as cut-off recompilation. We specify the language with an abstract, formal definition, parameterized over the two popular definitions of Standard ML: The 1997 Definition and the Harper-Stone interpretation.

1 Introduction

This document proposes a language extension of Standard ML to support separate compilation, the ability to divide a program into meaningful subparts that can be compiled in isolation and later linked to form a complete program. We support the cleaving of any ML program between two top-level declarations. Such a fragment is known as a "unit," which we give a unique name with global scope. Because these units are open code (they depend on the preceding code for typing and evaluation context), they do not make sense in isolation. The concept of an "assembly" is a collection of units with enough context to make them individually meaningful. Context for a unit is provided by "opening" other units. A unit can open another unit from two sources. First, it can open a unit that appears earlier in the same assembly. In this case, the interface for the opened unit can be computed by compiling the opened unit first. Second, a unit can be opened from another assembly. In this case, since the assembly must make sense on its own, this external dependency must be mediated by an interface that the programmer specifies. The first kind of open permits smooth scaling from existing SML code bases. Providing interfaces supports more flexible development patterns and permits parallel and cut-off compilation implementation strategies.

The language, if adopted, would allow for source-level compatibility of libraries and large programs between different implementations. Moreover, as a *language* rather than a *tool*, we are able to give a formal semantics to the extension that allows the programmer to understand his program's meaning in the abstract. It also allows implementors to have a robust way of answering questions such as, "when do certain changes to source files require recompilation of dependencies?"

The Standard ML language has two popular definitions. First is the *Definition of Standard ML* [MTHM97], published in 1997, whose semantics is given in terms of "semantic objects" (finite maps and sets). Most implementations use this definition. Harper and Stone also defined Standard ML via elaboration into a type-theoretic internal language, given in *A Type-Theoretic Interpretation of Standard ML* [HS00]. A few implementations, notably TILT [TIL05], use this semantics. The majority of real programs have the same meaning in both definitions. Because of these two different definitions, the language extension presented here is modularly defined in terms of a parsimonious interface to the underlying semantics. We give implementations of this interface for both the Definition (Appendix I) and Harper-Stone (Appendix H). Another consequence of this setup is that the separate compilation system is isolated from the language to the degree that it could be a starting point for extensions to languages other than SML.

1.1 Separate Compilation and Incremental Recompilation

Describing programs in the proposed language allows the resolution of compilation to be increased in two ways. We call these separate compilation and incremental recompilation.

Separate compilation is the practice of compiling assemblies (collections of units) in isolation. Because external dependencies must have their interfaces specified completely, an assembly is self-contained and the units within it can be compiled without the external dependencies present. Separate compilation allows programs to be developed in parallel pieces and then linked together at a later time.

The units in an assembly also permit incremental recompilation. Because a given program is often compiled many times, it is advantageous to have the compiler re-use the results of previous compilations whenever possible. One simple kind of recompilation avoids recompiling a unit whose dependencies have not changed since the previous compile. This notion of "have not changed" can be relaxed in order to permit reuse in more circumstances. For instance, if the programmer specifies an interface for a unit, then changes to that unit need not cause recompilation of units that open it.

As an abstract language specification, this document does not define how compilation and linking are performed; this varies widely between implementations. Instead we define the conditions under which a unit compiles and program links, and the meaning of a linked executable (its dynamic semantics). This makes it possible for implementors to reason about their implementations with respect to a definition, and to state and prove the correctness of their optimizations. (Indeed, the correctness of these optimizations is not trivial [WDE98, DWE98].)

Though the implementation of separate compilation and incremental recompilation is not part of the language proposal, part of the motivation for this language is to support these features in real compiler tools. Section 3 thus explains in more detail what compilation and linking might consist of for various ML implementation strategies. In Section 3.1 we additionally explain how TILT implements these features and how they behave from the user's perspective.

1.2 Design Decisions

Based on the number of different separate compilation systems for various SML implementations (and related languages such as O'Caml), it is safe to say that the space of possible designs is large. In this section we justify the design decisions made for our particular system.

Language-based. First and foremost, the design should be a *language* rather than a tool. By language, we mean a description of a set of valid programs, each of which is assigned a meaning. This definition should be formal and unambiguous. In particular, we wish to avoid the common situation where the description of the language is really documentation of a reference implementation.

Language-based solutions have many advantages. They give implementors an official answer to subtle questions, leading to a high degree of compatibility. They make it possible to reason abstractly about the system, which means that both the language implementors and language users can have confidence in their programs.

Environment independence. In keeping with our language-based design philosophy, the design must be isolated from its environment. For example, many project description languages force a unit to be stored in a file based on the name of the unit. Aside from making it more difficult to formally specify the language (as the specification would have to explain, say, the Posix filesystem), it may also make the language unimplementable. For instance, `///` is a valid structure identifier in SML, but not a valid Posix filename; case sensitivity differs between filesystems on popular platforms; and on Windows, it is impossible to create files with a base name of `con`, `aux`, and others. Though our system of course allows for units to be stored in files, it is the *tool* (compiler) implementing the language that mediates between the environment and the abstract language definition.

Completeness. We wish to allow an SML program to be cleaved into units between any two top-level declarations. This means that an interface should be able to describe any kind of kind of top-level binding in SML. Moreover, a unit should not be artificially restricted to contain exactly one declaration. This implies that we should not identify the concept of "unit" with that of SML structures, as is done in O'Caml. If we did so, then top-level value declarations, such as the ones required by the Standard Basis Library, could not be separately compiled against, for instance. Identifying units and structures in SML is even more untenable than it is in O'Caml—because SML does not allow functors or signatures inside structures, it would make it impossible to separately compile functors and signatures at all!

An unfortunate fact of Standard ML is that some structure expressions have no most general signature that can be written by the programmer (a consequence of the avoidance problem [GP92]). This means that some units cannot be ascribed an accurate interface by the programmer. For programs that incorporate such units, we can only obtain the appropriate context for the compilation of the rest of the program by compiling these units from source. This means that they must be part of the same assembly; in the terminology from Section 1.1, such units can be incrementally recompiled against but not separately compiled. As a consequence, our design must permit the omission of interfaces in order to accept all existing programs in a non-degenerate way. In practice however, instances of this problem are rare.

Simplicity. Although there is constant tension to add features of convenience or research interest to the project system, we strive for the simplest possible design. This is for several reasons. To encourage adoption, we want the project description language to be easy to implement. To encourage use by programmers, we want it to be useful but easy to understand. As a system that does the minimum required, it can serve as a common starting point for implementation-specific extensions. For example, we do not propose a standard mechanism for environment variables, conditional compilation, or compiler directives, leaving this for future work.

Conservativity. Along with the extension itself being simple, it should not affect the core or module languages in any substantial way. This precludes for instance "forward references" in assemblies, which would allow recursive dependencies that are not otherwise expressible in the language.

Automatic dependency analysis. We do not support automatic dependency analysis as is done in SML/NJ's Compilation Manager. For one, this would violate our desire for simplicity. It also would probably mean violating completeness, as dependency analysis systems like SML/NJ's generally make restrictions on the programs to exclude multiple top-level declarations with the same name, or even top-level declarations other than `structure`, `functor`, and `signature` bindings. Furthermore, these systems pose problems for language-based approaches in that they are too ambiguous: Dependencies on the order of effects are not at all evident from source code, so a program may have multiple legal topological sortings with different meaning. In contrast, our language is entirely unambiguous; assemblies have at most one meaning, and the order of unit evaluation is the same as the order they appear in the description.

Despite this, our language provides an appropriate target for the output of dependency analysis tools, and such tools are probably of substantial use.

Definite references. Our proposal is based on the idea that unit names are unique within a program; each is a *definite reference* to exactly one unit implementation. This is in contrast to functor arguments (for instance), which may have multiple instantiations, or structure bindings, which may shadow one another. The requirement for definite references is a necessary consequence of compiling code with free references [HP05]. The use of definite references also means that our formalism is much simpler than the type theory of modules. For example, use of our system does not introduce any sharing constraints whatsoever. Definite references also make implementation simpler: For instance, TILT mangles each unit identifier and simply uses Unix's `ld` to link separately compiled code.

The remainder of this document proceeds as follows. First, we give a high-level summary of the concepts and features of our proposal. Then, we walk through the abstract specification, with pointers into the interesting parts of the formalism (Section 2). The details of the Harper-Stone and Definition implementations are relegated to their own appendices (A and B). Programmers who prefer to see concrete syntax and code examples may wish to skip Section 2 entirely.

After the technical walkthrough, we discuss the different possible notions of compilation and linking for the wide variety of SML implementations (Section 3), followed by a brief tour of our implementation with examples. Section 4 discusses the concrete syntax and some issues of parsing. The bulk of the proposal is the formal specification of the language extension, which makes up the remainder (Appendices C–I).

1.3 Units, Interfaces, Assemblies, and Linking

For our proposal we introduce four new entities atop SML: *units*, *interfaces*, *assemblies*, and *programs*. Units and interfaces have already been discussed; these are the individual pieces of code that are compiled, and their descriptions for the purpose compiling in the absence of the implementation. Assemblies are the particles of linking. They consist of a series of interface and unit declarations. A collection of assemblies (called a program) can be linked together to form another assembly. A "complete" assembly can be made into a final executable.

These additional levels are not burdensome: Any existing SML program can be compiled by making it into a single unit inside a degenerate assembly consisting only of that unit, which will already be "complete." (However, most existing SML programs have a natural division into units based on the organization of code into files.)

Every unit declaration declares a unit identifier to have an interface (either implicit or explicit). This is used for checking and compiling the remainder of the assembly. A unit declaration may also provide an implementation for

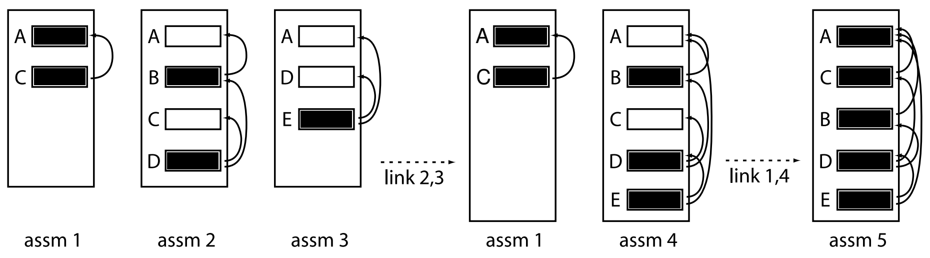

Figure 1: An example program being linked. The letters are unit names. Filled boxes correspond to unit declarations with implementations. The arrows represent the list of opened units (only units with implementations may open others). In the first step, three separate assemblies can be separately developed and the constituent units separately compiled. We can partially link assemblies 2 and 3 to give us a fourth assembly. This assembly still has unimplemented units, so it cannot be made into an executable yet. Linking it with assembly 1, however, resolves all of these dependencies and so an executable can be produced.

that unit. If it does not, then the declaration is an external dependency to be provided by another assembly during linking. Because each unit identifier is globally unique (a definite reference), we are able to make consistent reference to it in multiple assemblies, and ensure that all uses are talking about the same actual implementation. In a complete assembly, exactly one implementation for each unit must be provided. In order to link assemblies together, where one assembly implements a unit used to satisfy dependencies in other assemblies, all of the interfaces assigned to the unit must be equivalent.

Every unit (with implementation) declaration that relies on other units must make this list explicit by "opening" the units. The contents of these units are then available within the opening unit without qualification. Interfaces also specify an open list in the same way; note that in Standard ML, we have dependencies of interfaces on implementations, which is not the case for many other languages.

An example illustrating the linking of a few simple assemblies is given in Figure 1.

The order of units in an assembly is important. Only units declared earlier in the same assembly can be opened, to avoid recursive dependencies. The order that unit implementations appear is the order that their effects occur in the final completed program. Because this order matters, linking is also an asymmetric process: Linking assemblies 2 and 3 is successful in Figure 1, but linking units 3 and 2 would not be.

unit	A collection of SML top-level declarations with free references. Not generally meaningful without context
interface	Describes a unit; may also have free references to other units
assembly	Collection of units and interfaces that is individually meaningful. Refers to external units by giving their interfaces
open list	Explicit list of the units that a unit or interface depends on
EL	The external language for assemblies; the abstract syntax of what the programmer writes
IL	The internal language for assemblies
elaboration	Type-checking and transformation from EL to IL
linking	Creation of a single assembly from a series of assemblies
program	List of assemblies to be linked
completion	Finalization of an assembly that has no unimplemented units

Figure 2: A glossary of terms used in this proposal.

2 Technical Overview

This section discusses the technical details of the proposal. A glossary of terms is given in Figure 2 to serve as a reminder. The formalism given in Appendices C–I "speaks for itself," so we only cover the salient features in the English description. Also, we only describe the common interface to the two definitions here; the individual details of those are found in the appendix (A and B).

The separate compilation system is given in terms of two languages, an external language (EL) and an internal language (IL), the latter of which is parameterized on the underlying formalism (Harper-Stone or the Definition). A process of elaboration translates from EL to IL, type-checking the assembly as it does. Linking and evaluation are defined in terms of these IL phrases.

External Language. The EL is what the programmer writes down to describe an assembly. The abstract syntax is given in Table 3 (the concrete syntax is described in section 1). This includes the entire syntax of SML, used to form a unit implementation.

Here is a very simple example of an abstract assembly:

```
interface QUEUE =
   open in
    (structure Queue :
        sig
         type 'a queue
         val empty : 'a queue
         val push : 'a * 'a queue -> 'a queue
        end)

unit Q : QUEUE

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

The first line of the assembly binds the interface identifier `QUEUE` to an interface expression for a unit containing a structure `Queue`. An interface expression is a series of *topspecs*, which are anything that can appear in a signature, along with specs for functors and signature declarations. The next line declares the unit `Q` to have interface `QUEUE` but does not specify an implementation. Because the implementation of `Q` is not provided, the assembly is *incomplete* and cannot be linked into an executable alone. However, the declaration of `Q` makes it possible for the unit `C` to open it, and use it according to the interface supplied.

Here, the unit `C` is checked in a context with a single structure binding `Queue`. A unit `B` opening `A` achieves two things: in addition to making the bindings from `A` available to `B`, it indicates to the linking process that whenever `B` is linked into a program, `A` must also be linked. This is true even if `B` does not actually make use of `A`, perhaps because it relies only on top-level effects that `A` causes.

The declaration of unit `C` does not specify an interface, so the most general one is computed from the implementation. However, a declaration with no implementation (like the declaration of `Q`) must specify an interface, because there is no implementation from which to glean it.

Because interfaces can also depend on units (for their type components), an interface expression can additionally open units in the same manner. Here, `QUEUE` opens nothing.

These are essentially all of the features of the external language—the full abstract syntax is given in Table 3. An assembly is checked for well-formedness during the process of *elaboration*, which transforms EL code into IL code (or rejects the code as ill-formed).

Internal Language. The IL, whose syntax is given in Table 6, is even simpler than the EL. The main complication is that, because we support two different formalisms for the SML core, the IL is parameterized by a few syntactic classes and judgments. These appear in Table 5. For the syntax, the relevant parameters are the syntactic categories for compiled units and interfaces, *impl* and *intf*. Our previous example translates loosely into:

```
unit Q :
   (structure Queue :
       sig
        type 'a queue
        val empty : 'a queue
        val push : 'a * 'a queue -> 'a queue
       end)

unit C :
     (val q : int Q.Queue.queue
      val q' : int Q.Queue.queue) =
   require Q in
     (val q = Q.Queue.empty
      val q' = Q.Queue.push (0, q))
```

The main changes from the EL are as follows: First, we simply use interface expressions in place rather than naming them. Second, every unit declaration now comes explicitly with its interface. Since the declaration of `C` did not specify one in the EL example, the most general interface is computed as part of elaboration.

Unfortunately, the example given is neither here nor there; the code in parentheses would be the elaborated SML code and interfaces, whose format is determined by the parameter to the IL. For the loose example we simply did not carry out any compilation. Here, we have mimicked the Harper-Stone version in that we have made all references to other units via their unit identifiers (rather than implicitly as a result of being opened). The appearance of `Q` in the require clause for `C` simply records the fact that we did open `Q` in the EL. This will prevent us from discarding `Q`, which may be needed for its effects, even if it is not referenced in `C`.

Given the parameters, which tell us how to check and use IL units and interfaces, type-checking an IL assembly is straightforward. The entry point is the judgment $\mathit{pdecs} \vdash \mathit{assm}\ \mathsf{ok}$. *pdecs* is a context of unit identifiers and their interfaces. To check an assembly, we simply check each unit in sequence, given the *pdecs* produced by earlier unit declarations (Rules D.1.2 and D.1.3).

Elaboration. Elaboration produces an IL assembly from an EL assembly. This process is also parameterized by some operations from the underlying formalism, which are given in Table 10. The parameters are as follows: First, we need an interface for the top-level basis that can be assumed by every SML program (it defines types like `int` and exceptions like `Match`, among other things). This unit will be implicitly included in every assembly and opened for every unit. We also need a way to elaborate SML code (a unit) in some context, generating code and an interface for that code. We need to be able to compile EL interfaces to IL interfaces. Because the language gives the ability to *seal* units by ascribing interfaces to them, we additionally require an ascription operation that checks an IL unit against an IL interface and generates a new unit at that interface. (The new unit may discard components or hold certain types abstract, etc., so the code is in general different.)

In order to elaborate, we use an elaboration context $\mathcal{E}$ representing the unit and interface declarations that have been processed so far (Table 11).

The judgments in Table 12 describe the various steps in elaboration, with $\mathcal{E} \vdash \mathit{assembly} \leadsto \mathit{assm}$, the elaboration of entire assemblies, being the entry point.

For the most part, elaboration is a straightforward application of the parameters for elaborating units and interfaces. These themselves are unsurprising; see Sections H.2 and I.2 for the details.

Linking. A collection of assemblies, which we call a *program*, can be linked to form another assembly or a finished executable expression.

A program consists of a series of require and select directives. In an implementation, these would likely be given as command line arguments to the compiler. The require directive specifies that all the constituent units must be included in the linked result. The select directive allows for selective linking, so that only units from the assembly that are actually required by the rest of the program are included. The order of these directives in a program is important, and specifies the order of effects for the included units.

The linker is parameterized, as before (Table 14). We require a class of executable programs *prog*, and a judgment of their well-formedness. We also need to be able to query a unit or interface to see if it depends on a specific unit (to implement selective linking). Finally, given an assembly with no unresolved dependencies (except for the basis unit), a parameter $\vdash$ *assm* $\rightsquigarrow$ *prog* tells us how to convert this into an executable *prog*. This process is called completion.

The most interesting rules are the ones that carry out the require and select directives (Judgment F.2). These process each individual unit in an assembly in series. When we see a unit the first time in a require (Rule F.2.29), we simply include it in the resulting assembly. This first appearance may have an implementation; if it does not, then the resulting assembly will have unresolved dependencies and so cannot be completed without further linking. Subsequent occurrences (Rule F.2.30) must not have implementations. This is because each unit should have at most one implementation, and because we do not allow "forward references." For these occurrences, we simply check that the declared interface matches the existing one and continue.

The select directive is very similar. For the initial occurrence of a unit, the rule is split into two cases (Rules F.2.32 and F.2.33), based on whether the rest of the program uses the unit or not. If it is not used, then it is not included in the linked program. A program should contain at least one require directive, or no units will be included in the link!

The other interesting thing about linking is completion. An assembly is complete if it has no unresolved dependencies—unit declarations without an implementation—other than the basis unit. This is checked by the judgment *pdecs* $\vdash$ *assm* complete (Rules in F.5). The actual process of completion is entirely up to the underlying formalism.

Evaluation. We have specified the conditions under which programs can formed into executable expressions. The final step is to assign meaning to these programs in the form of a dynamic semantics. A single parameter (Table 18) $\vdash$ *prog* $\Rightarrow$ *res* specifies the result of a program, which is either term (successful termination) or raise (an uncaught exception). Here our attempt to give a single account of both the Definition and Harper-Stone falls somewhat flat: of course, the meaning of a program should be more than just this single bit of information, and moreover, many programs of interest do not fall into either category (because they do not terminate).

The problem has two sources: First, the Definition and Harper-Stone have entirely different notions of the results of evaluation, which makes comparing the abstract outcome of running a program quite difficult. Second, the Definition uses a "big-step" semantics, which makes it impossible to state anything about non-terminating programs.

To understand how evaluation works, it will be necessary to look at the specific instantiations (Sections H.4 and I.4). For Harper-Stone, evaluation is trivial (simply invoking the dynamic semantics on the linked expression), but for the Definition we build a superstructure above the dynamic semantics in order to maintain an environment of units.

3 Compilation and Linking in Practice

We have given an abstract language definition that uses words like "compilation" and "linking," but have not tied these to the traditional implementation concepts of compilation and linking. Although this is not the job of an abstract language definition, we give some practical thoughts on this issue here.

What are compilation and linking in our framework? The short answer is that compilation and linking can be almost anything that the implementor desires. The abstract nature of the specification confers this freedom. By implementing the Definition semantics directly, "compilation" essentially amounts to merely type-checking the input; the Definition version does not apply any interesting program transformations. Linking and completion are no-ops, and execution is interpretation. For an interpreter-based implementation of SML like HaMLet [ham], this may be precisely the correct choice: here, "separate compilation" means simply "separate type-checking."

A similar scenario may obtain for real compilers. For instance, MLton [MLt05], which is a whole-program compiler, cannot compile a source file to machine code without the rest of the program available. For MLton, the process of "separate compilation" amounts to "separate type-checking" as well. It is only when all of the parts of the program are "linked" to completion that the compiler can actually perform its whole-program analyses. When compiling is as simple as type-checking, there is not much benefit to supporting incremental recompilation.

For more traditional compilers, separate compilation has a more traditional meaning. For instance, TILT produces a `.o` file for each unit, along with a compiled version of the interface. Linking is performed using the system linker. Doing as much compilation as possible makes recompilation, and reuse of libraries (such as the Standard Basis) in multiple programs, more efficient. On the other hand, it may make it impossible to perform certain global optimizations like cross-unit inlining. Fortunately, there is no need to necessarily choose one or the other: For TILT, we plan to also provide a flag that compiles an entire complete source program as a single unit, in the case that we want to enable intra-unit optimizations for the entire program. In the following section we walk through a few examples using TILT, to give a feel for how a tool implementing the proposed language might work.

3.1 TILT Implementation and Examples

XXX TO-DO: here is where we appeal to hackers, by giving a small example that shows off the SC/IR features of TILT; its command-line syntax, etc. (I'll need Dave's help..)

unit identifiers	U		
open list	*opens*	::=	$\{U_1 \dots U_n\}$
interface identifiers	I		
interface expressions	*ie*	::=	`"`*filename*`"` $\langle$*opens*$\rangle$
declarations	*dec*	::=	`interface` $I = ie$
		$\mid$	`unit` U $\langle$: $I\rangle$ = `"`*filename*`"` $\langle$*opens*$\rangle$
		$\mid$	`unit` $U : I$

Table 1: EL concrete syntax

top-level specifications	*tspec*	::=	*spec*	(MTHM 14)
		$\mid$	`functor` *fspec*	
		$\mid$	`signature` *sigbind*	
functor specifications	*fspec*	::=	*funid*(*strid* : *sigexp*$_1$) : *sigexp*$_2$ $\langle$`and` *fspec*$\rangle$	
		$\mid$	`infix` $\langle d\rangle$ $vid_1 \cdots vid_n$	
		$\mid$	`infixr` $\langle d\rangle$ $vid_1 \cdots vid_n$	
		$\mid$	`nonfix` $vid_1 \cdots vid_n$	

Table 2: Interface concrete syntax. Interface files consist of a series of top-level specifications.

4 External Language Concrete Syntax

4.1 Parsing

We do not formalize the process of parsing in this document, because we are much more concerned with the static and dynamic semantics than the surface syntax. Parsing the concrete syntax (Tables 1 and 2) into the abstract syntax is simple, except for a few small issues.

First, the list of "opened" modules (given with curly braces) is optional for both interfaces and unit implementations. The meaning when they are present is obvious; when absent,[1] this should be parsed as the list of all unit declarations preceding this one in the file, in the order they appear. This gives programmers a convenient syntax for small programs, or when migrating codebases without explicit dependencies (*e.g.* SML/NJ CM files). Of course, providing an accurate "open" list generally results in better incremental recompilation and selective linking performance.

More complex is the issue of `infix` and related "fixity" declarations. Fixity declarations do not appear in the abstract syntax at all; they are an entirely parse-time construct. Although they may appear in both interfaces and units, no matching takes place at parse time—a unit that declares `infix x` matches an interface that declares `nonfix x` (or nothing at all), and vice versa, because neither declaration makes it to the elaboration phase. For the purposes of parsing later units, an implementation should use only the infix declarations in

[1] Note that the programmer represents an empty list of opens with empty curly braces.

the interfaces of the opened units. However, since an omitted interface indicates that the interface should be gleaned from the implementation, for code where no interface is supplied, the fixity context that the implementation produces should be used instead.

Finally, because the initial environment provides a fixity context, but the references to it are installed automatically by elaboration, the parser needs to also implicitly include this fixity context (before the fixity declarations for the opened units) when parsing each unit.

Syntactic compatibility between implementations is of course important. Because this issue is not trivial, if this proposal is accepted with this concrete syntax, it would be worthwhile to formalize the parsing process. Better still, dealing with infix in a more principled way (which would probably require changes to the SML core) that makes parsing obvious might be possible.

5 Conclusion

We have presented a modest extension to Standard ML for the organization of large projects spread across multiple compliation units. The system permits separate compilation (where the notions of compilation and linking necessarily varies from compiler to compiler) and optimizations such as incremental recompilation. Importantly, the proposal is in the form of a language with a formal specification. In order to simultaneously target multiple definitions of the source language, the extension is given in terms of an abstract interface, which we implement with both the original Definition and the type-theoretic Harper-Stone interpretation.

Although many extensions are possible, and desirable, we feel that this proposal represents a minimal useful extension that should be easy to implement and understand by programmers. We hope that the proposed language can serve as a testbed for such future extensions while providing a well-defined and stable subset for the purpose of sharing code between implementations.

References

[DWE98] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is java binary compatibility? In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 341–361, 1998.

[GP92] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing, 1992. Circulated in manuscript form. Full version in *Theoretical Computer Science*, 193(1–2):75–96, February 1998.

[ham] HaMLet web site), year = 2005, note = URL: http://www.ps.uni-sb.de/hamlet/, key = HaMLet.

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

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

[MLt05] MLton web site, 2005. URL: http://mlton.org/.

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

[TIL05] TILT web site, 2005. URL: http://www.tilt.cs.cmu.edu/.

[WDE98] D. Wragg, S. Drossopoulou, and S. Eisenbach. Java binary compatibility is almost correct. Technical Report 3/98, Imperial College, 1998. http://www-dse/projects/SLURP/bc.

A The Harper-Stone Interpretation

This section describes the instantiation of our proposal with the Harper-Stone formalism [HS00]. Because it is aimed at implementors, knowledge of HS is a prerequisite.

The Internal Language. The HS IL supports almost all of the features necessary to implement the IL, so we need only add a few small things. The syntax is given in Table 20. An *impl* (unit implementation) is just an HS *mod* (module). In the HS IL, modules can contain functors, so this is everything that we need. An interface naturally has a collection of *sdecs* (signature declarations) to describe the components of the module. However, because signatures do not appear in modules or as *sdecs*, we also include a series of *tdec*s. These are declarations *of* signatures. These signatures may depend on the components of the module, so the module is bound to a variable; the form of an interface thus is *var* : [*sdecs*]; *tdecs*.

To check that a unit matches an interface (Rule H.1.3.45), we just use the HS notion of signature matching to check that *mod* has type [*sdecs*]. The *tdecs* are an entirely an interface artifact, so they are merely checked for well-formedness.

Interface equivalence is interesting because of the way it treats the bound variable (Rule H.1.4.46). After checking that the two *sdecs* are the same, we create a *selfified* version of the first *sdecs*. This means that abstract types are made transparently equal to projections from the bound variable *var*; for instance, the signature `sig type t end` becomes `sig type t =` *var*`.t end`. We then check that the two *tdecs* are equal in a context where *var* has the signature [*sdecs*] and *var'* has the selfified signature. This ensures that projections of type components from *var* and *var'* will be treated as equal.

The other interesting rule in the IL is context creation (Rule H.1.8.52). This is used by the IL to convert a series of *pdecs* (*unitid* : *intf*) to a context for type-checking a unit or interface. For each *pdec*, we generate the corresponding HS IL variable $\overline{unitid}$, which is simply bound in the context with the corresponding signature [*sdecs*].

Elaboration. Elaboration (Section H.2) is by far the most involved process in the HS version.

The elaboration of unit expressions is a good starting point (Rule H.2.1.53). It begins by using the list of imported units to create a context *udecs* and a substitution σ.

Context preparation is done with Rule H.2.11.70, which is really the heart of elaboration. The units in the context *pdecs* are used to form an HS IL context *decs* via IL context creation (Rule H.1.8.52). An HS IL context consists of entries of the form *lab* $\triangleright$ *var* : *con*, which binds the variable *var* at the type *con*, along with the label *lab*. The label is used to resolve references from the external language. The units in the first part of the context, $udecs_0$, are not accessible from the external language, so they are all given the label 1 arbitrarily—this label cannot be written down in the external language, and in fact will never be referred to. Note that every unit in scope goes in the context at this point. For each unit in the import list we then produce its selfified signature (as above), sig_i, based on the HS IL variable that corresponds to the unit identifier. We bind this signature in the context, using the variable bound in its interface (which saves us from having to substitute for it to match) and the label $1^\star$. Although, as before, the programmer cannot refer to this module because he cannot write the label 1, the "star convention" of context lookup during HS elaboration means that the module is treated as "open," so that identifier lookups descend into the module. Thus, though the programmer cannot refer to an imported unit directly, he can use its contents without qualification. The path that results from lookup is a projection from the HS IL variable var_i. Finally, we prepare a substitution σ. A unit elaborated in this context will make references to the imports through the variables var_i, but we will want these to instead refer to the actual implementation, which will be bound to variable $\overline{unitid_i}$.

With the context and substitution prepared, we can return to Rule H.2.1.53 for elaborating a unit expression. To do so, we elaborate the unit body *topdec* to a series of top-level bindings along with corresponding specs (*sdecs*; *tdecs*). The substitution σ is applied all around. Because the bindings are collected together into a module (written using square brackets in HS), we need to patch up the top-level *sdecs* and *tdecs* as well. This is accomplished by Rule H.2.3.55, which just puts the *sdecs* (value, structure, functor specs, etc.) inside a signature and then adjusts the *tdecs* (signature declarations that may depend on those sdecs) to project through the bound module variable.

Another interesting rule is the one for interface ascription (Rule H.2.4.56. Recall that this coerces a unit to a supplied interface, producing a new unit expression. To do this we use the HS coercion compilation judgment, which works on a path to the original module (in this case, var_0), producing a new module expression mod'. The coercion compilation judgment also produces this new module's most general signature (*sig*), which may have more type equations than the interface we are matching against. To produce the resulting module, we need to bind the input module to a variable and evaluate mod'; this is done with an in-place functor application. Because the signature is the most general one, in the sense that it may expose more type equations than we asked for, we additionally coerce this module to [*sdecs*]. We also check that the old set of *tdecs* is a superset of the new set. This check doesn't affect the new unit we produce, since *tdecs* (declarations of signatures) are a construct only of the interfaces.

Linking. Linking is very simple in the HS interpretation. A complete program is just an expression of type unit,[2] which consists of a series of bindings for the constituent units inside a single module (Rules in H.3.1 and H.3.2). We simply project the last component—which is always the empty record—from this module, in order to ensure that the entire expression has unit type. Type-checking is

[2]As in the empty record type, not compilation units!

just HS type-checking (Rule H.3.5.82), and a unit or interface requires another unit iff it is in its free variable set (Rules H.3.3.80 and H.3.4.81).

Evaluation. Because we have prepared an HS expression by the process of completion, evaluation is completely trivial. We simply invoke the HS dynamic semantics in the empty environment and return either term or raise as appropriate (Rules in H.4.1).

B The Definition

This section describes the instantiation of our proposal with the original Definition formalism [MTHM97].

Because the Definition essentially type-checks and evaluates the external language directly, this instantiation is mainly concerned with establishing a superstructure above the static and dynamic semantics to check and evaluate the assembly structure, by extending the semantic objects to account for these new features.

The Internal Language. For the Definition, the IL is almost the same as the EL (Table 35). A unit implementation is just an EL unit expression, perhaps coerced to a series of interfaces. An interface consists of finite maps F, G, and E, mapping functor identifiers, signature identifiers, and structure/type/value identifiers to the appropriate static semantic objects, respectively. To support definite references for separate compilation, each interface also has a set of imports IP and exports EP, that attach labels to what would otherwise be alpha-varying variables. These labels are just natural numbers. To see how this works, consider Rule I.1.2.86, which type-checks a unit implementation. After preparing the basis B from the imports, we type-check the *topdec* using the judgment from the Definition, which produces a new basis B'. The "clos" operation (Table 39) then creates the interface for this unit from the new basis and the import environment out of Γ. The maps F, G, and E come directly from the basis, but clos must generate the imports and exports. Exports is simple: the "exp" operation simply generates a natural number label for each exported type name, which is the set T. The imports list IP binds a set of type names (the ones used in the implementation, but not the ones it exports) to projections of labels from unit identifiers ($unitid.n$). This is simply the inverse of the import environment IE that the implementation was checked in. Thus, the basis produced is "closed" by this import environment, making reference to other units only through these projections.

The "inst" function, also in Table 39, is the opposite operation: it instantiates a closed interface to produce a Basis. This is used in the creation of an Definition basis from a set of unit imports (Rule I.1.6.92), which was also used above to type-check a unit. We start with B_0, the set of type names that appear in the context's import environment. The rest of the bases may refer to these. Then, each unit's interface is instantiated in Γ using the inst operation.

Inst looks up the unit's interface in the environment (1st clause), ensures that there are no name collisions (which can always be achieved by alpha-varying; 2nd clause) and then instantiates the components of the basis F, G and E (3rd clause). This is done by looking up each type name bound in the imports list inside the import environment, and substituting through the phrase.

Elaboration. Given the setup, elaboration is quite easy (Section I.2). Unlike the Harper-Stone formalism, elaboration is only a type-checking process, so we leave the implementations untouched.

Linking. Linking in the Definition formalism is completely trivial (Section I.3, because a program is just an assembly.

Evaluation. In the Harper-Stone implementation, a linked program was just an expression to evaluate, so evaluation was trivial. For the Definition, the process is slightly more involved, because a linked program is just an IL assembly, and unit implementations are just EL units. So, rather than invoke the dynamic semantics directly, we explain how to extend the Definition dynamic semantics to also evaluate units and assemblies. Still, this process is quite straightforward.

The main extension is the unit environment, UE, which binds Definition bases to unit identifiers. To evaluate an assembly, we simply evaluate units in sequence. An uncaught exception packet results in early termination of the assembly (Rules I.4.2.114 and I.4.2.115). Otherwise (Rule I.4.2.113), we simply add the dynamic basis produced from evaluating the unit to the unit environment, and evaluate the remainder of the assembly.

Individual units are evaluated according to the Definition. The only interesting rule is the one for evaluating an implementation coerced to an interface (Rule I.4.3.118). Here we must thin the basis produced from the implementation with the $\downarrow$ operation, which is defined as a straightforward extension of the Definition operation in Table 47.

C External Language for Assemblies

assembly	::=	$\cdot$	
		assembly, *intid* = *intexp*	interface definition
		assembly, *unitid* : *intexp*	unit description
		assembly, *unitid* $\langle$: *intexp*$\rangle$ = *unitexp*	unit definition
unitexp	::=	open $unitid_1 \cdots unitid_n$ in *topdec*	(MTHM 14)
intexp	::=	open $unitid_1 \cdots unitid_n$ in *topspec*	
		intid	
topspec	::=	*spec*	(MTHM 14)
		functor *funspec*	functor
		signature *sigbind*	signature
		$topspec_1$ $topspec_2$	
funspec	::=	*funid*(*strid* : *sigexp*) : $sigexp'$ $\langle$and *funspec*$\rangle$	

Table 3: EL syntax

No *topspec* or *funspec* may describe the same identifier twice.

Table 4: EL syntactic restrictions

D Internal Language for Assemblies

intf	compilation unit interface
impl	compilation unit implementation
Γ	context
$\Gamma \vdash \mathit{intf} : \mathsf{Intf}$	*intf* is well-formed
$\Gamma \vdash \mathit{impl} : \mathit{intf}$	*impl* has interface *intf*
$\Gamma \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}$	interface equivalence
$\Gamma \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}$	*intf* is a sub-interface of intf'
$\mathit{pdecs} \vdash \Gamma$	Γ declares units in *pdecs* (q.v.)
$\vdash \Gamma\ \mathsf{ok}$	Γ is well-formed

Table 5: IL parameters

assm	::=	$\cdot$	
		$\mathit{assm}, \mathit{unitid} : \mathit{intf}$	unit description
		$\mathit{assm}, \mathit{unitid} : \mathit{intf} = \mathit{unite}$	unit definition
unite	::=	$\mathsf{require}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{impl}$	
pdecs	::=	$\cdot$	
		$\mathit{pdecs}, \mathit{pdec}$	
pdec	::=	$\mathit{unitid} : \mathit{intf}$	unit description

Table 6: IL syntax

Section	*Judgement...*	*Meaning ...*
D.1	$\mathit{pdecs} \vdash \mathit{assm}\ \mathsf{ok}$	*assm* is well-formed
D.2	$\mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf}$	*intf* is well-formed
D.3	$\mathit{pdecs} \vdash \mathit{unite} : \mathit{intf}$	*unite* has interface *intf*
D.4	$\mathit{pdecs} \vdash \mathit{impl} : \mathit{intf}$	*impl* has interface *intf*
D.5	$\mathit{pdecs} \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}$	interface equivalence
D.6	$\mathit{pdecs} \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}$	*intf* is a sub-interface of intf'
D.7	$\vdash \mathit{pdecs}\ \mathsf{ok}$	*pdecs* is well-formed

Table 7: IL static semantics

The syntactic categories *assm* and *pdecs* specify lists of elements.

- We denote by $(\cdot, \cdot)$ the operations of syntactic concatenation for them; for example, $\mathit{assm}, \mathit{assm}'$.
- We sometimes work at the front of lists as if they were built up from left to right; for example, $\mathit{pdec}, \mathit{pdecs}$.
- We sometimes omit the initial $\cdot$; for example, $\mathit{pdec}_1, \ldots, \mathit{pdec}_n$.

Table 8: IL notation

Function	*Definition*		
$\mathrm{dom}(\mathit{pdecs})$	$\mathrm{dom}(\mathit{pdec}_1, \ldots, \mathit{pdec}_n)$	$=$	$\{\mathrm{dom}(\mathit{pdec}_1), \ldots, \mathrm{dom}(\mathit{pdec}_n)\}$
$\mathrm{dom}(\mathit{pdec})$	$\mathrm{dom}(\mathit{unitid} : \mathit{intf})$	$=$	unitid

Table 9: IL dom($\cdot$)

D.1 Assemblies $\boxed{\mathit{pdecs} \vdash \mathit{assm}\ \mathsf{ok}}$

$$\frac{\vdash \mathit{pdecs}\ \mathsf{ok}}{\mathit{pdecs} \vdash \cdot\ \mathsf{ok}} \tag{1}$$

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

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

D.2 Interfaces $\boxed{\mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf}}$

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

D.3 Unit Expressions $\boxed{\mathit{pdecs} \vdash \mathit{unite} : \mathit{intf}}$

$$\frac{\begin{array}{c}\mathit{unite} = \mathsf{require}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{impl} \\ \mathit{unitid}_1 \in \mathrm{dom}(\mathit{pdecs}) \quad \cdots \quad \mathit{unitid}_n \in \mathrm{dom}(\mathit{pdecs}) \\ \mathit{pdecs} \vdash \mathit{impl} : \mathit{intf}\end{array}}{\mathit{pdecs} \vdash \mathit{unite} : \mathit{intf}} \tag{5}$$

D.4 Implementations $\boxed{\mathit{pdecs} \vdash \mathit{impl} : \mathit{intf}}$

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

D.5 Interface Equivalence $\boxed{\mathit{pdecs} \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}}$

$$\frac{\begin{array}{c}\mathit{pdecs} \vdash \Gamma \\ \Gamma \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}\end{array}}{\mathit{pdecs} \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}} \tag{7}$$

D.6 Sub-interface Relation $\boxed{\mathit{pdecs} \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}}$

$$\frac{\begin{array}{c}\mathit{pdecs} \vdash \Gamma \\ \Gamma \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}\end{array}}{\mathit{pdecs} \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}} \tag{8}$$

D.7 Assembly Declaration Lists $\boxed{\vdash \mathit{pdecs}\ \mathsf{ok}}$

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

$$\frac{\begin{array}{c}\vdash \mathit{pdecs}\ \mathsf{ok} \\ \mathit{unitid} \notin \mathrm{dom}(\mathit{pdecs}) \\ \mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf}\end{array}}{\vdash \mathit{pdecs}, \mathit{unitid} : \mathit{intf}\ \mathsf{ok}} \tag{10}$$

D.8 Properties of the Internal Language

We assume that the parameters to the IL static semantics satisfy the following lemma. Informally, we require that if a judgement holds, then its constituent parts are well-formed.

Lemma 1 (Parameter Well-formedness) *The following propositions hold:*

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

If a judgement in the IL static semantics holds, then its constituent parts are well-formed.

Lemma 2 (Well-formedness) *The following propositions hold:*

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

E Elaboration

$\mathit{intf}_{\mathit{basis}}$	basis interface
$\mathit{pdecs} \vdash \mathit{unitexp} \rightsquigarrow \mathit{impl} : \mathit{intf}$	unit expressions
$\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf}$	interface expressions
$\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{impl}$	interface ascription

Table 10: Elaborator parameters

$\mathcal{E}$	$::=$	$\cdot$	
		$\mathcal{E}, \mathit{pdec}$	unit description
		$\mathcal{E}, \mathit{intid} : \mathsf{Intf} = \mathit{intf}$	interface definition

Table 11: Elaborator syntax

Section	*Judgement...*	*Meaning ...*
E.1	$\vdash \mathit{assembly} \rightsquigarrow \mathit{assm}; \mathcal{E}$	assemblies
E.2	$\mathcal{E} \vdash \mathit{unitexp} \rightsquigarrow \mathit{unite} : \mathit{intf}$	unit expressions
E.3	$\mathcal{E} \vdash \mathit{intexp} \rightsquigarrow \mathit{intf} : \mathsf{Intf}$	interface expressions
E.4	$\mathcal{E} \vdash \mathit{unite}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{unite}$	interface ascription
E.5	$\mathcal{E} \vdash \mathit{pdecs}$	*pdecs* declares units in $\mathcal{E}$
E.6	$\vdash \mathcal{E}\ \mathsf{ok}$	$\mathcal{E}$ is well-formed

Table 12: Elaborator judgements

Function	*Definition*		
$\mathrm{dom}(\mathcal{E})$	$\mathrm{dom}(\cdot)$	$=$	$\emptyset$
	$\mathrm{dom}(\mathcal{E}, \mathit{pdec})$	$=$	$\mathrm{dom}(\mathcal{E}) \cup \{\mathrm{dom}(\mathit{pdec})\}$
	$\mathrm{dom}(\mathcal{E}, \mathit{intid} : \mathsf{Intf} = \mathit{intf})$	$=$	$\mathrm{dom}(\mathcal{E}) \cup \{\mathit{intid}\}$

Table 13: Elaborator dom(·)

E.1 Assemblies $\boxed{\vdash \textit{assembly} \rightsquigarrow \textit{assm}; \mathcal{E}}$

$$\frac{}{\vdash \cdot \rightsquigarrow \textit{basis} : \textit{intf}_{\textit{basis}}; \textit{basis} : \textit{intf}_{\textit{basis}}} \quad (11)$$

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

$$\frac{\begin{array}{c}\vdash \textit{assembly} \rightsquigarrow \textit{assm}; \mathcal{E} \\ \textit{intid} \notin \mathrm{dom}(\mathcal{E}) \\ \mathcal{E} \vdash \textit{intexp} \rightsquigarrow \textit{intf}\end{array}}{\vdash \textit{assembly}, \textit{intid} = \textit{intexp} \rightsquigarrow \textit{assm}; \mathcal{E}, \textit{intid} : \mathsf{Intf} = \textit{intf}} \quad (12)$$

$$\frac{\begin{array}{c}\vdash \textit{assembly} \rightsquigarrow \textit{assm}; \mathcal{E} \\ \textit{unitid} \notin \mathrm{dom}(\mathcal{E}) \\ \mathcal{E} \vdash \textit{intexp} \rightsquigarrow \textit{intf}\end{array}}{\vdash \textit{assembly}, \textit{unitid} : \textit{intexp} \rightsquigarrow \textit{assm}, \textit{unitid} : \textit{intf}; \mathcal{E}, \textit{unitid} : \textit{intf}} \quad (13)$$

$$\frac{\begin{array}{c}\vdash \textit{assembly} \rightsquigarrow \textit{assm}; \mathcal{E} \\ \textit{unitid} \notin \mathrm{dom}(\mathcal{E}) \\ \mathcal{E} \vdash \textit{unitexp} \rightsquigarrow \textit{unite} : \textit{intf}\end{array}}{\begin{array}{l}\vdash \textit{assembly}, \textit{unitid} = \textit{unitexp} \rightsquigarrow \\ \quad \textit{assm}, \textit{unitid} : \textit{intf} = \textit{unite}; \mathcal{E}, \textit{unitid} : \textit{intf}\end{array}} \quad (14)$$

$$\frac{\begin{array}{c}\vdash \textit{assembly} \rightsquigarrow \textit{assm}; \mathcal{E} \\ \textit{unitid} \notin \mathrm{dom}(\mathcal{E}) \\ \mathcal{E} \vdash \textit{intexp} \rightsquigarrow \textit{intf} \\ \mathcal{E} \vdash \textit{unitexp} \rightsquigarrow \textit{unite}_0 : \textit{intf}_0 \\ \mathcal{E} \vdash \textit{unite}_0 : \textit{intf}_0 \preceq \textit{intf} \rightsquigarrow \textit{unite}\end{array}}{\begin{array}{l}\vdash \textit{assembly}, \textit{unitid} : \textit{intexp} = \textit{unitexp} \rightsquigarrow \\ \quad \textit{assm}, \textit{unitid} : \textit{intf} = \textit{unite}; \mathcal{E}, \textit{unitid} : \textit{intf}\end{array}} \quad (15)$$

E.2 Unit Expressions $\boxed{\mathcal{E} \vdash \textit{unitexp} \rightsquigarrow \textit{unite} : \textit{intf}}$

$$\frac{\begin{array}{c}\mathcal{E} \vdash \textit{pdecs} \\ \textit{unitexp} = \mathsf{open}\ \textit{unitid}_1 \cdots \textit{unitid}_n\ \mathsf{in}\ \textit{topdec} \\ \textit{pdecs} \vdash \mathsf{open}\ \textit{basis}\ \textit{unitid}_1 \cdots \textit{unitid}_n\ \mathsf{in}\ \textit{topdec} \rightsquigarrow \textit{impl} : \textit{intf} \\ \textit{unite} = \mathsf{require}\ \textit{basis}\ \textit{unitid}_1 \cdots \textit{unitid}_n\ \mathsf{in}\ \textit{impl}\end{array}}{\mathcal{E} \vdash \textit{unitexp} \rightsquigarrow \textit{unite} : \textit{intf}} \quad (16)$$

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

E.3 Interface Expressions $\boxed{\mathcal{E} \vdash \mathit{intexp} \rightsquigarrow \mathit{intf} : \mathsf{Intf}}$

$$\frac{\mathcal{E} \vdash \mathit{pdecs} \quad \mathit{pdecs} \vdash \mathsf{open}\ \mathit{basis}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf}}{\mathcal{E} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf} : \mathsf{Intf}} \quad (17)$$

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

$$\frac{\mathcal{E} = \mathcal{E}', \mathit{intid} : \mathsf{Intf} = \mathit{intf}, \mathcal{E}''}{\mathcal{E} \vdash \mathit{intid} \rightsquigarrow \mathit{intf} : \mathsf{Intf}} \quad (18)$$

E.4 Interface Ascription $\boxed{\mathcal{E} \vdash \mathit{unite}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{unite}}$

$$\frac{\begin{array}{c}\mathit{unite}_0 = \mathsf{require}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{impl}_0 \\ \mathcal{E} \vdash \mathit{pdecs} \quad \mathit{pdecs} \vdash \Gamma \quad \Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{impl} \\ \mathit{unite} = \mathsf{require}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{impl}\end{array}}{\mathcal{E} \vdash \mathit{unite}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{unite}} \quad (19)$$

E.5 Context Coercion $\boxed{\mathcal{E} \vdash \mathit{pdecs}}$

$$\frac{}{\cdot \vdash \cdot} \quad (20)$$

$$\frac{\mathcal{E} \vdash \mathit{pdecs}}{\mathcal{E}, \mathit{unitid} : \mathit{intf} \vdash \mathit{pdecs}, \mathit{unitid} : \mathit{intf}} \quad (21)$$

$$\frac{\mathcal{E} \vdash \mathit{pdecs}}{\mathcal{E}, \mathit{intid} : \mathsf{Intf} = \mathit{intf} \vdash \mathit{pdecs}} \quad (22)$$

E.6 Well-formed Elaboration Contexts $\boxed{\vdash \mathcal{E}\ \mathsf{ok}}$

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \quad (23)$$

$$\frac{\vdash \mathcal{E}\ \mathsf{ok} \quad \mathit{unitid} \notin \mathrm{dom}(\mathcal{E}) \quad \mathcal{E} \vdash \mathit{pdecs} \quad \mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf}}{\vdash \mathcal{E}, \mathit{unitid} : \mathit{intf}\ \mathsf{ok}} \quad (24)$$

$$\frac{\vdash \mathcal{E}\ \mathsf{ok} \quad \mathit{intid} \notin \mathrm{dom}(\mathcal{E}) \quad \mathcal{E} \vdash \mathit{pdecs} \quad \mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf}}{\vdash \mathcal{E}, \mathit{intid} : \mathsf{Intf} = \mathit{intf}\ \mathsf{ok}} \quad (25)$$

E.7 Properties of the Elaborator

The elaborator's translation judgements have the form

$$\textit{context} \vdash \textit{input} \rightsquigarrow \textit{output}.$$

The main property of the elaborator is that if *context* is well-formed and such a judgement holds, then *output* is well-formed.

We assume that the parameters to the elaborator satisfy the following lemma.

Lemma 3 (Parameter Well-formed Translation) *The following propositions hold:*

1. *If* $\cdot \vdash \textit{intf}_{\textit{basis}} : \mathsf{Intf}$.
2. *If* $\textit{pdecs} \vdash \textit{unitexp} \rightsquigarrow \textit{impl} : \textit{intf}$ *and* $\vdash \textit{pdecs}\ \mathsf{ok}$*, then* $\textit{pdecs} \vdash \textit{impl} : \textit{intf}$.
3. *If* $\textit{pdecs} \vdash \mathsf{open}\ \textit{unitid}_1 \cdots \textit{unitid}_n\ \mathsf{in}\ \textit{topspec} \rightsquigarrow \textit{intf}$ *and* $\vdash \textit{pdecs}\ \mathsf{ok}$*, then* $\textit{pdecs} \vdash \textit{intf} : \mathsf{Intf}$.
4. *If* $\Gamma \vdash \textit{impl}_0 : \textit{intf}_0 \preceq \textit{intf} \rightsquigarrow \textit{impl}$*, then* $\Gamma \vdash \textit{impl}_0 : \textit{intf}_0$*,* $\Gamma \vdash \textit{intf}_0 \leq \textit{intf} : \mathsf{Intf}$*, and* $\Gamma \vdash \textit{impl} : \textit{intf}$.

Lemma 4 (Well-formed Translation) *The following propositions hold:*

1. *If* $\vdash \textit{assembly} \rightsquigarrow \textit{assm}; \mathcal{E}$*, then* $\cdot \vdash \textit{assm}\ \mathsf{ok}$ *and* $\vdash \mathcal{E}\ \mathsf{ok}$.
2. *If* $\vdash \mathcal{E}\ \mathsf{ok}$*, then* $\mathcal{E} \vdash \textit{pdecs}$*,* $\vdash \textit{pdecs}\ \mathsf{ok}$*, and*
   (a) *If* $\mathcal{E} \vdash \textit{unitexp} \rightsquigarrow \textit{unite} : \textit{intf}$*, then* $\textit{pdecs} \vdash \textit{unite} : \textit{intf}$.
   (b) *If* $\mathcal{E} \vdash \textit{intexp} \rightsquigarrow \textit{intf} : \mathsf{Intf}$*, then* $\textit{pdecs} \vdash \textit{intf} : \mathsf{Intf}$.
   (c) *If* $\mathcal{E} \vdash \textit{unite}_0 : \textit{intf}_0 \preceq \textit{intf} \rightsquigarrow \textit{unite}$*, then* $\textit{pdecs} \vdash \textit{unite}_0 : \textit{intf}_0$*,* $\textit{pdecs} \vdash \textit{intf}_0 \leq \textit{intf} : \mathsf{Intf}$*, and* $\textit{pdecs} \vdash \textit{unite} : \textit{intf}$.

F Linking

$prog$	linked programs
$\vdash prog$ ok	$prog$ is well-formed
$\vdash intf$ $\mathsf{requires}$ $unitid$	$intf$ depends on $unitid$
$\vdash impl$ $\mathsf{requires}$ $unitid$	$impl$ depends on $unitid$
$\vdash assm \rightsquigarrow prog$	completion

Table 14: Linker parameters

$program$	$::=$	$\cdot$	empty
		$program$; $\mathsf{require}$ $assm$	link
		$program$; select $assm$	link selectively

Table 15: Linking language

F.1	$\vdash program \rightsquigarrow prog$	completion
F.2	$pdecs \vdash program \rightsquigarrow assm$	linking
F.3	$\vdash assm$ $\mathsf{requires}$ $unitid$	$assm$ depends on $unitid$
F.4	$\vdash unite$ $\mathsf{requires}$ $unitid$	$unite$ depends on $unitid$
F.5	$pdecs \vdash assm$ $\mathsf{complete}$	$assm$ is complete

Table 16: Linker judgements

The conventions for lists (Table 8) apply to programs *program*.

Table 17: Linker conventions

F.1 Completion $\boxed{\vdash \mathit{program} \leadsto \mathit{prog}}$

$$\frac{\cdot \vdash \mathit{program} \leadsto \mathit{assm} \quad \cdot \vdash \mathit{assm}\ \mathsf{complete} \quad \vdash \mathit{assm} \leadsto \mathit{prog}}{\vdash \mathit{program} \leadsto \mathit{prog}} \tag{26}$$

F.2 Linking $\boxed{\mathit{pdecs} \vdash \mathit{program} \leadsto \mathit{assm}}$

$$\frac{\vdash \mathit{pdecs}\ \mathsf{ok}}{\mathit{pdecs} \vdash \cdot \leadsto \cdot} \tag{27}$$

$$\frac{\mathit{pdecs} \vdash \mathit{program} \leadsto \mathit{assm}}{\mathit{pdecs} \vdash \mathsf{require}\ \cdot; \mathit{program} \leadsto \mathit{assm}} \tag{28}$$

$$\frac{\begin{array}{c}\mathit{unitid} \notin \mathrm{dom}(\mathit{pdecs}) \\ \mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf} \quad \langle \mathit{pdecs} \vdash \mathit{unite} : \mathit{intf} \rangle \\ \mathit{pdecs}, \mathit{unitid} : \mathit{intf} \vdash \mathsf{require}\ \mathit{assm}'; \mathit{program} \leadsto \mathit{assm}\end{array}}{\begin{array}{l}\mathit{pdecs} \vdash \mathsf{require}\ (\mathit{unitid} : \mathit{intf}\langle = \mathit{unite}\rangle, \mathit{assm}'); \mathit{program} \leadsto \\ \quad \mathit{unitid} : \mathit{intf}\langle = \mathit{unite}\rangle, \mathit{assm}\end{array}} \tag{29}$$

$$\frac{\begin{array}{c}\mathit{pdecs} = \mathit{pdecs}', \mathit{unitid} : \mathit{intf}', \mathit{pdecs}'' \\ \mathit{pdecs} \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf} \\ \mathit{pdecs} \vdash \mathsf{require}\ \mathit{assm}'; \mathit{program} \leadsto \mathit{assm}\end{array}}{\mathit{pdecs} \vdash \mathsf{require}\ (\mathit{unitid} : \mathit{intf}, \mathit{assm}'); \mathit{program} \leadsto \mathit{assm}} \tag{30}$$

$$\frac{\mathit{pdecs} \vdash \mathit{program} \leadsto \mathit{assm}}{\mathit{pdecs} \vdash \mathsf{select}\ \cdot; \mathit{program} \leadsto \mathit{assm}} \tag{31}$$

$$\frac{\begin{array}{c}\mathit{unitid} \notin \mathrm{dom}(\mathit{pdecs}) \\ \mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf} \quad \langle \mathit{pdecs} \vdash \mathit{unite} : \mathit{intf} \rangle \\ \mathit{pdecs}, \mathit{unitid} : \mathit{intf} \vdash \mathsf{select}\ \mathit{assm}'; \mathit{program} \leadsto \mathit{assm} \\ \vdash \mathit{assm}\ \mathsf{requires}\ \mathit{unitid}\end{array}}{\begin{array}{l}\mathit{pdecs} \vdash \mathsf{select}\ (\mathit{unitid} : \mathit{intf}\langle = \mathit{unite}\rangle, \mathit{assm}'); \mathit{program} \leadsto \\ \quad \mathit{unitid} : \mathit{intf}\langle = \mathit{unite}\rangle, \mathit{assm}\end{array}} \tag{32}$$

$$\frac{\begin{array}{c}\mathit{unitid} \notin \mathrm{dom}(\mathit{pdecs}) \\ \mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf} \quad \langle \mathit{pdecs} \vdash \mathit{unite} : \mathit{intf} \rangle \\ \mathit{pdecs}, \mathit{unitid} : \mathit{intf} \vdash \mathsf{select}\ \mathit{assm}'; \mathit{program} \leadsto \mathit{assm} \\ \nvdash \mathit{assm}\ \mathsf{requires}\ \mathit{unitid}\end{array}}{\mathit{pdecs} \vdash \mathsf{select}\ (\mathit{unitid} : \mathit{intf}\langle = \mathit{unite}\rangle, \mathit{assm}'); \mathit{program} \leadsto \mathit{assm}} \tag{33}$$

$$\frac{\begin{array}{c} \mathit{pdecs} = \mathit{pdecs}', \mathit{unitid} : \mathit{intf}', \mathit{pdecs}'' \\ \mathit{pdecs} \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf} \\ \mathit{pdecs} \vdash \mathsf{select}\ \mathit{assm}'; \mathit{program} \rightsquigarrow \mathit{assm} \end{array}}{\mathit{pdecs} \vdash \mathsf{select}\ (\mathit{unitid} : \mathit{intf}, \mathit{assm}'); \mathit{program} \rightsquigarrow \mathit{assm}} \qquad (34)$$

F.3 Assembly Dependencies

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

$$\frac{\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}}{\vdash \mathit{unitid}' : \mathit{intf} \langle = \mathit{unite} \rangle, \mathit{assm}\ \mathsf{requires}\ \mathit{unitid}} \qquad (35)$$

$$\frac{\vdash \mathit{unite}\ \mathsf{requires}\ \mathit{unitid}}{\vdash \mathit{unitid}' : \mathit{intf} = \mathit{unite}, \mathit{assm}\ \mathsf{requires}\ \mathit{unitid}} \qquad (36)$$

$$\frac{\mathit{unitid}' \neq \mathit{unitid} \qquad \vdash \mathit{assm}\ \mathsf{requires}\ \mathit{unitid}}{\vdash \mathit{unitid}' : \mathit{intf} \langle = \mathit{unite} \rangle, \mathit{assm}\ \mathsf{requires}\ \mathit{unitid}} \qquad (37)$$

F.4 Unit Dependencies

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

$$\frac{\begin{array}{c} \mathit{unite} = \mathsf{require}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{impl} \\ \mathit{unitid} \in \{\mathit{unitid}_1, \ldots, \mathit{unitid}_n\} \end{array}}{\vdash \mathit{unite}\ \mathsf{requires}\ \mathit{unitid}} \qquad (38)$$

$$\frac{\begin{array}{c} \mathit{unite} = \mathsf{require}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{impl} \\ \vdash \mathit{impl}\ \mathsf{requires}\ \mathit{unitid} \end{array}}{\vdash \mathit{unite}\ \mathsf{requires}\ \mathit{unitid}} \qquad (39)$$

F.5 Complete Assemblies

$\boxed{\mathit{pdecs} \vdash \mathit{assm}\ \mathsf{complete}}$

$$\frac{\vdash \mathit{pdecs}\ \mathsf{ok}}{\mathit{pdecs} \vdash \cdot\ \mathsf{complete}} \qquad (40)$$

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

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

$$\frac{\begin{array}{c} \mathit{unitid} \notin \mathrm{dom}(\mathit{pdecs}) \\ \mathit{pdecs} \vdash \mathit{unite} : \mathit{intf} \\ \mathit{pdecs}, \mathit{unitid} : \mathit{intf} \vdash \mathit{assm}\ \mathsf{complete} \end{array}}{\mathit{pdecs} \vdash \mathit{unitid} : \mathit{intf} = \mathit{unite}, \mathit{assm}\ \mathsf{complete}} \qquad (42)$$

F.6 Properties of the Linker

We assume that the parameters to the linker satisfy the following lemma.

Lemma 5 (Linker Parameter Requirements) *The following propositions hold:*

1. *If* $\vdash \mathit{intf}$ requires *unitid and pdecs* $\vdash \mathit{intf} : \mathsf{Intf}$*, then* $\mathit{unitid} \in \mathrm{dom}(\mathit{pdecs})$.
2. *If* $\vdash \mathit{impl}$ requires *unitid and pdecs* $\vdash \mathit{impl} : \mathit{intf}$*, then* $\mathit{unitid} \in \mathrm{dom}(\mathit{pdecs})$.
3. *If* $\vdash \mathit{assm} \leadsto \mathit{prog}$ *and* $\cdot \vdash \mathit{assm}$ complete*, then* $\vdash \mathit{prog}$ ok.

The linker satsifies the following lemma.

Lemma 6 (Linker Well-formed Translation) *The following propositions hold:*

1. *If* $\vdash \mathit{program} \leadsto \mathit{prog}$*, then* $\vdash \mathit{prog}$ ok.
2. *If* $\mathit{pdecs} \vdash \mathit{program} \leadsto \mathit{assm}$*, and* $\vdash \mathit{pdecs}$ ok*, then* $\mathit{pdecs} \vdash \mathit{assm}$ ok.
3. *If* $\vdash \mathit{assm}$ requires *unitid and pdecs* $\vdash \mathit{assm}$ ok*, then* $\mathit{unitid} \in \mathrm{dom}(\mathit{pdecs})$.
4. *If* $\vdash \mathit{unite}$ requires *unitid and pdecs* $\vdash \mathit{unite} : \mathit{intf}$*, then* $\mathit{unitid} \in \mathrm{dom}(\mathit{pdecs})$.
5. *If* $\mathit{pdecs} \vdash \mathit{assm}$ complete*, then* $\mathit{pdecs} \vdash \mathit{assm}$ ok.

G Evaluation

$\vdash prog \Rightarrow res$	*prog* evaluates to *res* (q.v.)

Table 18: Evaluator parameters

res	::=	term	termination
		raise	uncaught exception

Table 19: Evaluator syntax

H The Harper-Stone Interpretation of Standard ML

H.1 Parameters for the Internal Language

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

Table 20: IL syntax (HS)

Section	*Judgement...*	*Meaning ...*
H.1.1	$\mathit{decs} \vdash \mathit{intf} : \mathsf{Intf}$	intf is well-formed
H.1.2	$\mathit{decs} \vdash \mathit{tdecs}\ \mathsf{ok}$	tdecs is well-formed
H.1.3	$\mathit{decs} \vdash \mathit{impl} : \mathit{intf}$	impl has interface intf
H.1.4	$\mathit{decs} \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}$	interface equivalence
H.1.5	$\mathit{decs} \vdash \mathit{tdecs} \equiv \mathit{tdecs}'$	tdecs equivalence
H.1.6	$\mathit{decs} \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}$	sub-interface relation
H.1.7	$\mathit{decs} \vdash \mathit{tdecs} \leq \mathit{tdecs}'$	tdecs inclusion
H.1.8	$\mathit{pdecs} \vdash \mathit{decs}$	decs declares units in pdecs
HS 3.4.1	$\vdash \mathit{decs}\ \mathsf{ok}$	decs is well-formed

Table 21: IL static semantics (HS)

Phrase	*Bound Variables*	*Scope*
$\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}$	var	tdecs

Table 22: HS bound variables and scopes

- We assume an injective function $\overline{\cdot}$ taking unit identifiers to HS IL variables and that there are countably many variables not in the range of this function.
- The syntactic category *tdecs* specifies a list of elements. The conventions in Table 8 apply to it.

Table 23: HS notation for the static semantics

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

Table 24: HS dom(·)

H.1.1 Well-formed Interfaces $\boxed{\mathit{decs} \vdash \mathit{intf} : \mathsf{Intf}}$

$$\frac{\mathit{var} \notin \mathrm{BV}(\mathit{decs}) \quad \mathit{decs} \vdash \mathit{sdecs}\ \mathsf{ok} \quad \mathit{decs}, \mathit{var} : [\mathit{sdecs}] \vdash \mathit{tdecs}\ \mathsf{ok}}{\mathit{decs} \vdash (\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}) : \mathsf{Intf}} \tag{43}$$

H.1.2 Well-formed Top-level Declaration Lists $\boxed{\mathit{decs} \vdash \mathit{tdecs}\ \mathsf{ok}}$

$$\frac{\begin{array}{c}\vdash \mathit{decs}\ \mathsf{ok} \\ \mathit{tdecs} = \mathit{sigid}_1 : \mathsf{Sig} = \mathit{sig}_1, \ldots, \mathit{sigid}_n : \mathsf{Sig} = \mathit{sig}_n \\ \mathit{sigid}_1, \ldots, \mathit{sigid}_n \text{ are distinct} \\ \mathit{decs} \vdash \mathit{sig}_1 : \mathsf{Sig} \quad \cdots \quad \mathit{decs} \vdash \mathit{sig}_n : \mathsf{Sig}\end{array}}{\mathit{decs} \vdash \mathit{tdecs}\ \mathsf{ok}} \tag{44}$$

H.1.3 Well-formed Implementations $\boxed{\mathit{decs} \vdash \mathit{impl} : \mathit{intf}}$

$$\frac{\mathit{var} \notin \mathrm{BV}(\mathit{decs}) \quad \mathit{decs} \vdash \mathit{mod} : [\mathit{sdecs}] \quad \mathit{decs}, \mathit{var} : [\mathit{sdecs}] \vdash \mathit{tdecs}\ \mathsf{ok}}{\mathit{decs} \vdash \mathit{mod} : (\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs})} \tag{45}$$

H.1.4 Interface Equivalence $\boxed{\mathit{decs} \vdash \mathit{intf} \equiv \mathit{intf}' : \mathsf{Intf}}$

$$\frac{\begin{array}{c}\mathit{var} \notin \mathrm{BV}(\mathit{decs}) \quad \mathit{var}' \notin \mathrm{BV}(\mathit{decs}) \cup \{\mathit{var}\} \\ \mathit{decs} \vdash \mathit{sdecs} \equiv \mathit{sdecs}' \quad \mathit{decs}, \mathit{var} : [\mathit{sdecs}] \vdash \mathit{var} : \mathit{sig} \\ \mathit{decs}, \mathit{var} : [\mathit{sdecs}], \mathit{var}' : \mathit{sig} \vdash \mathit{tdecs} \equiv \mathit{tdecs}'\end{array}}{\mathit{decs} \vdash (\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}) \equiv (\mathit{var}' : [\mathit{sdecs}']; \mathit{tdecs}') : \mathsf{Intf}} \tag{46}$$

Rule 46: If $\mathit{decs}, \mathit{var} : [\mathit{sdecs}] \vdash \mathit{var} : \mathit{sig}$ holds, then sig is a selfified signature for var; in particular, sig is fully transparent, maximizing type sharing when signatures in tdecs and tdecs' are compared.

H.1.5 *tdecs* equivalence $\boxed{\mathit{decs} \vdash \mathit{tdecs} \equiv \mathit{tdecs}'}$

$$\frac{\mathit{decs} \vdash \mathit{tdecs} \leq \mathit{tdecs}' \quad \mathit{decs} \vdash \mathit{tdecs}' \leq \mathit{tdecs}}{\mathit{decs} \vdash \mathit{tdecs} \equiv \mathit{tdecs}'} \tag{47}$$

H.1.6 Sub-interface Relation $\boxed{\mathit{decs} \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}}$

$$\frac{\begin{array}{c}\mathit{var} \notin \mathrm{BV}(\mathit{decs}) \quad \mathit{var}' \notin \mathrm{BV}(\mathit{decs}) \cup \{\mathit{var}\} \\ \mathit{decs} \vdash \mathit{sdecs} \leq \mathit{sdecs}' \quad \mathit{decs}, \mathit{var} : [\mathit{sdecs}] \vdash \mathit{var} : \mathit{sig} \\ \mathit{decs}, \mathit{var} : [\mathit{sdecs}], \mathit{var}' : \mathit{sig} \vdash \mathit{tdecs} \leq \mathit{tdecs}'\end{array}}{\mathit{decs} \vdash (\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}) \leq (\mathit{var}' : [\mathit{sdecs}']; \mathit{tdecs}') : \mathsf{Intf}} \tag{48}$$

Rule 48: If $\mathit{decs}, \mathit{var} : [\mathit{sdecs}] \vdash \mathit{var} : \mathit{sig}$ holds, then sig is a selfified signature for var; in particular, sig is fully transparent, maximizing type sharing when signatures in tdecs and tdecs' are compared.

H.1.7 *tdecs* Inclusion

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

$$\frac{decs \vdash tdecs \ \mathsf{ok}}{decs \vdash tdecs \leq \cdot} \tag{49}$$

$$\frac{\begin{array}{c} decs \vdash tdecs \leq tdecs' \\ sigid \notin \mathrm{dom}(tdecs') \\ tdecs = tdecs_1, sigid = sig' : \mathsf{Sig}, tdecs_2 \\ decs \vdash sig \equiv sig' : \mathsf{Sig} \end{array}}{decs \vdash tdecs \leq tdecs', sigid = sig : \mathsf{Sig}} \tag{50}$$

H.1.8 Context Creation

$$\boxed{pdecs \vdash decs}$$

$$\frac{}{\cdot \vdash \cdot} \tag{51}$$

$$\frac{\begin{array}{c} pdecs \vdash \Gamma \quad \overline{unitid} \notin \mathrm{BV}(\Gamma) \quad var \notin \mathrm{BV}(\Gamma) \\ \Gamma \vdash sdecs \ \mathsf{ok} \quad \Gamma, var : [sdecs] \vdash tdecs \ \mathsf{ok} \end{array}}{pdecs, unitid : (var : [sdecs]; tdecs) \vdash \Gamma, \overline{unitid} : [sdecs]} \tag{52}$$

H.1.9 Properties of the HS Parameters for the IL

Lemma 7 (Well-formedness) *The following propositions hold:*

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

H.2 Parameters for the Elaborator

udecs	$::=$	$\cdot$
		$\mathit{udecs}, \mathit{udec}$
udec	$::=$	sdec
		tdec
σ	$::=$	$\cdot$
		$\sigma, \mathit{var}/\mathit{var}'$
		$\sigma, \mathit{lab}.\mathit{var}/\mathit{var}'$

Table 25: HS elaborator syntax

Section	*Judgement…*	*Meaning …*
H.2.1	$\mathit{pdecs} \vdash \mathit{unitexp} \rightsquigarrow \mathit{impl} : \mathit{intf}$	unit expressions
H.2.2	$\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf} : \mathsf{Intf}$	interface expressions
H.2.3	$\mathit{udecs} \vdash \mathit{sdecs}; \mathit{tdecs} \rightsquigarrow \mathit{intf} : \mathsf{Intf}$	interface creation
H.2.4	$\mathit{decs} \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{impl}$	interface ascription
H.2.5	$\mathit{udecs} \vdash \mathit{topdec} \rightsquigarrow \mathit{sbnds} : (\mathit{sdecs}; \mathit{tdecs})$	top-level declarations
H.2.6	$\mathit{udecs} \vdash \mathit{topspec} \rightsquigarrow \mathit{sdecs}; \mathit{tdecs}$	top-level specifications
H.2.7	$\mathit{udecs} \vdash \mathit{sigbind} \rightsquigarrow \mathit{tdecs}$	signature bindings
H.2.8	$\mathit{udecs} \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig}$	signature expressions
H.2.9	$\mathit{udecs} \vdash \mathit{funspec} \rightsquigarrow \mathit{sdecs}$	functor specifications
H.2.10	$\mathit{udecs} \vdash_{\mathsf{ctx}} \mathit{sigid} \rightsquigarrow \mathit{sig} : \mathsf{Sig}$	signature lookup
H.2.11	$\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \rightsquigarrow \mathit{udecs}, \sigma$	*udecs* declares units in *pdecs* and top-level identifiers in unitid_i
H.2.12	$\vdash \mathit{udecs}\ \mathsf{ok}$	*udecs* is well-formed
H.2.13	$\mathit{udecs} \vdash \mathit{decs}$	*decs* declares constructors, modules, and expressions in *udecs*

Table 26: HS elaborator judgments

Function	*Definition*		
$\mathrm{dom}(\mathit{udecs})$	$\mathrm{dom}(\mathit{udec}_1, \ldots, \mathit{udec}_n)$	$=$	$\{\mathrm{dom}(\mathit{udec}_1), \ldots, \mathrm{dom}(\mathit{udec}_n)\}$
$\mathrm{dom}(\mathit{udec})$	$\mathrm{dom}(\mathit{sdec})$	$=$	$\mathrm{dom}(\mathit{sdec})$
	$\mathrm{dom}(\mathit{tdec})$	$=$	$\mathrm{dom}(\mathit{tdec})$
$\mathrm{dom}(\mathit{sdec})$	$\mathrm{dom}(\mathit{lab} \triangleright \mathit{dec})$	$=$	lab
$\mathrm{BV}(\mathit{sdec})$	$\mathrm{BV}(\mathit{lab} \triangleright \mathit{dec})$	$=$	$\mathrm{BV}(\mathit{dec})$

Table 27: HS elaborator dom(·) and BV(·)

Phrase	*Bound Variables*	*Scope*
sdec, *udecs*	BV(*sdec*)	*udecs*

Table 28: HS elaborator bound variables and scopes

- The basis interface is defined by $\mathit{intf}_{\mathit{basis}} = \mathit{var} : \mathit{sig}_{\mathit{basis}}; \cdot$.
- The HS elaborator assumes the presence of a structure $\overline{\mathit{basis}}{:}\mathit{sig}_{\mathit{basis}}$ serving as the initial basis for the HS IL. It must contain at least the following fields which define three exceptions:

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

  This assumption is satisfied by the basis unit.

- An HS elaboration context *udecs* is a list of structure and top-level declarations; this is an extension to HS where elaboration contexts are structure declaration lists (*sdecs*) that may contain duplicate labels.
- The HS external language supports higher-level functors but Standard ML does not. For compatiblity with SML, we modify the HS EL and elaborator as follows.
  - Remove functor *funbind* from the syntax of EL structure declarations (HS 34) and Rule 205 (HS 47) for elaborating htem.
  - Remove functor $\mathit{funid}(\mathit{strid} : \mathit{sigexp}) : \mathit{sigexp}'$ from the syntax of EL structure specifications and Rule 224 (HS 50) for elaborating them.

  These changes, together with Rules 59 and 66, ensure that EL functors may only be defined at the top-level of a compilation unit.

- The conventions for lists (Table 8) apply to HS elaboration contexts *udecs* and to substitution lists σ.
- The notation $\{\mathit{phrase}/\mathit{var}'\}\mathit{phrase}$, denotes the capture-free substitution of *phrase* for free occurrences of *var* within phrase'.

Table 29: HS elaborator conventions

$$\begin{array}{lcl}\{\cdot\}\mathit{phrase} & = & \mathit{phrase}\\ \{\sigma, \mathit{var}/\mathit{var}'\}\mathit{phrase} & = & \{\mathit{var}/\mathit{var}'\}(\{\sigma\}\mathit{phrase})\\ \{\sigma, \mathit{lab}.\mathit{var}/\mathit{var}'\}\mathit{phrase} & = & \{\mathit{lab}.\mathit{var}/\mathit{var}'\}(\{\sigma\}\mathit{phrase})\end{array}$$

Table 30: HS elaborator substitution $\{\sigma\}\mathit{phrase}$

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

Table 31: HS elaborator shadowing $tdecs +\!\!+ \, tdecs'$

H.2.1 Unit Expressions

$\boxed{\mathit{pdecs} \vdash \mathit{unitexp} \rightsquigarrow \mathit{impl} : \mathit{intf}}$

$$\frac{\begin{array}{c}\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \rightsquigarrow \mathit{udecs}, \sigma \\ \mathit{udecs} \vdash \mathit{topdec} \rightsquigarrow \mathit{sbnds} : (\mathit{sdecs}; \mathit{tdecs}) \\ \mathit{impl} = [\{\sigma\}\mathit{sbnds}] \quad \mathit{udecs} \vdash \{\sigma\}\mathit{sdecs}; \{\sigma\}\mathit{tdecs} \rightsquigarrow \mathit{intf} : \mathsf{Intf}\end{array}}{\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topdec} \rightsquigarrow \mathit{impl} : \mathit{intf}} \tag{53}$$

H.2.2 Interface Expressions

$\boxed{\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf} : \mathsf{Intf}}$

$$\frac{\begin{array}{c}\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \rightsquigarrow \mathit{udecs}, \sigma \\ \mathit{udecs} \vdash \mathit{topspec} \rightsquigarrow \mathit{sdecs}; \mathit{tdecs} \\ \mathit{udecs} \vdash \{\sigma\}\mathit{sdecs}; \{\sigma\}\mathit{tdecs} \rightsquigarrow \mathit{intf} : \mathsf{Intf}\end{array}}{\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf} : \mathsf{Intf}} \tag{54}$$

H.2.3 Interface Creation

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

$$\frac{\begin{array}{c}\mathit{var} \notin \mathrm{BV}(\mathit{decs}) \\ \mathit{sdecs} = \mathit{lab}_1 \triangleright \mathit{dec}_1, \ldots, \mathit{lab}_n \triangleright \mathit{dec}_n \\ \mathit{var}_1 = \mathrm{BV}(\mathit{dec}_1) \quad \cdots \quad \mathit{var}_n = \mathrm{BV}(\mathit{dec}_n) \\ \sigma = \mathit{var}.\mathit{lab}_1/\mathit{var}_1, \ldots, \mathit{var}.\mathit{lab}_n/\mathit{var}_n \\ \mathit{intf} = (\mathit{var} : [\mathit{sdecs}]; \{\sigma\}\mathit{tdecs})\end{array}}{\mathit{udecs} \vdash \mathit{sdecs}; \mathit{tdecs} \rightsquigarrow \mathit{intf} : \mathsf{Intf}} \tag{55}$$

H.2.4 Interface Ascription

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

$$\frac{\begin{array}{c}\mathit{var}_0 \neq \mathit{var} \\ \mathit{decs}, \mathit{var}_0 : [\mathit{sdecs}_0] \vdash_{\mathsf{sub}} \mathit{var}_0 : [\mathit{sdecs}_0] \preceq [\mathit{sdecs}] \rightsquigarrow \mathit{mod}' : \mathit{sig} \\ \mathit{decs}, \mathit{var}_0 : [\mathit{sdecs}_0], \mathit{var} : \mathit{sig} \vdash \mathit{tdecs} \leq \mathit{tdecs}_0 \\ \mathit{mod} = (((\lambda \mathit{var}_0 : [\mathit{sdecs}_0].\mathit{mod}')\ \mathit{mod}_0) : [\mathit{sdecs}])\end{array}}{\mathit{decs} \vdash \mathit{mod}_0 : (\mathit{var}_0 : [\mathit{sdecs}_0]; \mathit{tdecs}_0) \preceq (\mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}) \rightsquigarrow \mathit{mod}} \tag{56}$$

Rule 56: The rules for the coercion compiler ensure that *sig* is fully transparent, maximizing type sharing when signatures in *tdecs* and tdecs_0 are compared.

H.2.5 Top-Level Declarations

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

$$\frac{\begin{array}{c}\mathit{udecs} \vdash \mathit{strdec} \rightsquigarrow \mathit{sbnds} : \mathit{sdecs} \\ \langle \mathit{udecs}, \mathit{sdecs} \vdash \mathit{topdec} \rightsquigarrow \mathit{sbnds}' : (\mathit{sdecs}'; \mathit{tdecs})\rangle\end{array}}{\begin{array}{l}\mathit{udecs} \vdash \mathit{strdec}\ \langle \mathit{topdec}\rangle \rightsquigarrow \\ \quad \mathit{sbnds}\langle +\!\!+ \mathit{sbnds}'\rangle : (\mathit{sdecs}\langle +\!\!+ \mathit{sdecs}'\rangle; \cdot\langle +\!\!+ \mathit{tdecs}\rangle)\end{array}} \tag{57}$$

$$\frac{\begin{array}{c} \mathit{udecs} \vdash \mathit{sigbind} \leadsto \mathit{tdecs} \\ \langle \mathit{udecs}, \mathit{tdecs} \vdash \mathit{topdec} \leadsto \mathit{sbnds} : (\mathit{sdecs}; \mathit{tdecs}') \rangle \end{array}}{\begin{array}{l} \mathit{udecs} \vdash \textsf{signature}\ \mathit{sigbind}\ \langle \mathit{topdec} \rangle \leadsto \\ \quad \cdot\langle, \mathit{sbnds}\rangle : (\cdot\langle, \mathit{sdecs}\rangle; \mathit{tdecs}\langle +\!\!+ \mathit{tdecs}'\rangle) \end{array}} \tag{58}$$

$$\frac{\begin{array}{c} \mathit{udecs} \vdash \mathit{funbind} \leadsto \mathit{sbnds} : \mathit{sdecs} \\ \langle \mathit{udecs}, \mathit{sdecs} \vdash \mathit{topdec} \leadsto \mathit{sbnds}' : (\mathit{sdecs}'; \mathit{tdecs}) \rangle \end{array}}{\begin{array}{l} \mathit{udecs} \vdash \textsf{functor}\ \mathit{funbind}\ \langle \mathit{topdec} \rangle \leadsto \\ \quad \mathit{sbnds}\langle +\!\!+ \mathit{sbnds}'\rangle : (\mathit{sdecs}\langle +\!\!+ \mathit{sdecs}'\rangle; \cdot\langle +\!\!+ \mathit{tdecs}\rangle) \end{array}} \tag{59}$$

H.2.6 Top-Level Specifications

$\boxed{\mathit{udecs} \vdash \mathit{topspec} \leadsto \mathit{sdecs}; \mathit{tdecs}}$

$$\frac{\mathit{udecs} \vdash \mathit{spec} \leadsto \mathit{sdecs}}{\mathit{udecs} \vdash \mathit{spec} \leadsto \mathit{sdecs}; \cdot} \tag{60}$$

$$\frac{\mathit{udecs} \vdash \mathit{funspec} \leadsto \mathit{sdecs}}{\mathit{udecs} \vdash \textsf{functor}\ \mathit{funspec} \leadsto \mathit{sdecs}; \cdot} \tag{61}$$

$$\frac{\mathit{udecs} \vdash \mathit{sigbind} \leadsto \mathit{tdecs}}{\mathit{udecs} \vdash \textsf{signature}\ \mathit{sigbind} \leadsto \cdot; \mathit{tdecs}} \tag{62}$$

$$\frac{\begin{array}{c} \mathit{udecs} \vdash \mathit{topspec}_1 \leadsto \mathit{sdecs}_1; \mathit{tdecs}_1 \\ \mathit{udecs}, \mathit{sdecs}_1, \mathit{tdecs}_1 \vdash \mathit{topspec}_2 \leadsto \mathit{sdecs}_2; \mathit{tdecs}_2 \\ \mathit{udecs} \vdash \mathit{decs} \\ \mathit{decs} \vdash \mathit{sdecs}_1, \mathit{sdecs}_2\ \textsf{ok} \quad \mathit{decs} \vdash \mathit{tdecs}_1, \mathit{tdecs}_2\ \textsf{ok} \end{array}}{\mathit{udecs} \vdash \mathit{topspec}_1\ \mathit{topspec}_2 \leadsto \mathit{sdecs}_1, \mathit{sdecs}_2; \mathit{tdecs}_1, \mathit{tdecs}_2} \tag{63}$$

Rule 63: Because of include, there is no way to restrict the syntax to ensure that the concatenations $\mathit{sdecs}_1, \mathit{sdecs}_2$ and $\mathit{tdecs}_1, \mathit{tdecs}_2$ are well-formed.

H.2.7 Signature Bindings

$\boxed{\mathit{udecs} \vdash \mathit{sigbind} \leadsto \mathit{tdecs}}$

$$\frac{\begin{array}{c} \mathit{udecs} \vdash \mathit{sigexp} \leadsto \mathit{sig} : \textsf{Sig} \\ \langle \mathit{udecs} \vdash \mathit{sigbind} \leadsto \mathit{tdecs} \quad \mathit{sigid} \notin \mathrm{dom}(\mathit{tdecs}) \rangle \end{array}}{\mathit{udecs} \vdash \mathit{sigid} = \mathit{sigexp}\ \langle \textsf{and}\ \mathit{sigbind} \rangle \leadsto \mathit{sigid} = \mathit{sig}\langle, \mathit{tdecs}\rangle} \tag{64}$$

H.2.8 Signature Expressions

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

To the rules in HS Section 6.6.8, we add the following rule for signature definitions.

$$\frac{\mathit{udecs} \vdash_{\textsf{ctx}} \mathit{sigid} \leadsto \mathit{sig} : \textsf{Sig}}{\mathit{udecs} \vdash \mathit{sigid} \leadsto \mathit{sig} : \textsf{Sig}} \tag{65}$$

H.2.9 Functor Specifications

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

$$\frac{\begin{array}{c}\mathit{udecs} \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \quad \mathit{var} \notin \mathrm{BV}(\mathit{udecs}) \\ \mathit{udecs}, \overline{\mathit{strid}} \rhd \mathit{var} : \mathit{sig} \vdash \mathit{sigexp}' \leadsto \mathit{sig}' : \mathsf{Sig} \\ \langle \mathit{udecs} \vdash \mathit{funspec} \leadsto \mathit{sdecs} \quad \overline{\mathit{funid}} \notin \mathrm{dom}(\mathit{sdecs})\rangle\end{array}}{\begin{array}{l}\mathit{udecs} \vdash \mathit{funid}(\mathit{strid} : \mathit{sigexp}) : \mathit{sigexp}' \ \langle \mathsf{and}\ \mathit{funspec}\rangle \leadsto \\ \quad \overline{\mathit{funid}} : (\mathit{var} : \mathit{sig} \rightharpoonup \mathit{sig}')\langle, \mathit{sdecs}\rangle\end{array}} \tag{66}$$

H.2.10 Signature Lookup

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

$$\frac{}{\mathit{udecs}, \mathit{sigid} : \mathsf{Sig} = \mathit{sig} \vdash_{\mathsf{ctx}} \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}} \tag{67}$$

$$\frac{\mathit{sigid}' \neq \mathit{sigid} \quad \mathit{udecs} \vdash_{\mathsf{ctx}} \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}}{\mathit{udecs}, \mathit{sigid}' : \mathsf{Sig} = \mathit{sig}' \vdash_{\mathsf{ctx}} \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}} \tag{68}$$

$$\frac{\mathit{udecs} \vdash_{\mathsf{ctx}} \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}}{\mathit{udecs}, \mathit{sdec} \vdash_{\mathsf{ctx}} \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}} \tag{69}$$

H.2.11 Context Creation

$\boxed{\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \leadsto \mathit{udecs}, \sigma}$

$$\frac{\begin{array}{c}\mathit{unitid}_1, \ldots, \mathit{unitid}_n \in \mathrm{dom}(\mathit{pdecs}) \\ \mathit{pdecs} \vdash \mathit{decs} \quad \mathit{decs} = \mathit{dec}_1, \ldots, \mathit{dec}_m \\ \mathit{udecs}_0 = 1 \rhd \mathit{dec}_1, \ldots, 1 \rhd \mathit{dec}_m \\ \mathit{pdecs} = \mathit{pdecs}'_1, \mathit{unitid}_1 : (\mathit{var}_1 : [\mathit{sdecs}_1]; \mathit{tdecs}_1), \mathit{pdecs}''_1 \\ \mathit{decs} \vdash \overline{\mathit{unitid}_1} : \mathit{sig}_1 \quad \mathit{udecs}_1 = 1^\star \rhd \mathit{var}_1 : \mathit{sig}_1, \mathit{tdecs}_1 \\ \vdots \\ \mathit{pdecs} = \mathit{pdecs}'_n, \mathit{unitid}_n : (\mathit{var}_n : [\mathit{sdecs}_n]; \mathit{tdecs}_n), \mathit{pdecs}''_n \\ \mathit{decs} \vdash \overline{\mathit{unitid}_n} : \mathit{sig}_n \quad \mathit{udecs}_n = 1^\star \rhd \mathit{var}_n : \mathit{sig}_n, \mathit{tdecs}_n \\ \mathit{udecs} = \mathit{udecs}_0, \mathit{udecs}_1, \ldots, \mathit{udecs}_n \\ \sigma = \overline{\mathit{unitid}_1}/\mathit{var}_1, \ldots, \overline{\mathit{unitid}_n}/\mathit{var}_n\end{array}}{\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \leadsto \mathit{udecs}, \sigma} \tag{70}$$

Rule 70: If $\mathit{decs} \vdash \overline{\mathit{unitid}_i} : \mathit{sig}_i$ holds, then sig_i is a selfified signature for $\overline{\mathit{unitid}_i}$.

H.2.12 Well-formed Contexts

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

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \tag{71}$$

$$\frac{\vdash \mathit{udecs}\ \mathsf{ok} \quad \mathit{udecs} \vdash \mathit{decs} \quad \mathit{decs} \vdash \mathit{dec}\ \mathsf{ok}}{\vdash \mathit{udecs}, \mathit{lab} \rhd \mathit{dec}\ \mathsf{ok}} \tag{72}$$

$$\frac{\vdash \mathit{udecs}\ \mathsf{ok} \quad \mathit{udecs} \vdash \mathit{decs} \quad \mathit{decs} \vdash \mathit{tdec}\ \mathsf{ok}}{\vdash \mathit{udecs}, \mathit{tdec}\ \mathsf{ok}} \tag{73}$$

H.2.13 Context Coercion $\boxed{\mathit{udecs} \vdash \mathit{decs}}$

$$\frac{\mathit{udecs} \vdash \mathit{decs}}{\mathit{udecs}, \mathit{lab} \rhd \mathit{dec} \vdash \mathit{decs}, \mathit{dec}} \tag{74}$$

$$\frac{\mathit{udecs} \vdash \mathit{decs}}{\mathit{udecs}, \mathit{tdec} \vdash \mathit{decs}} \tag{75}$$

H.2.14 Properties of the HS Parameters for the Elaborator

Lemma 8 (Well-formed Translation) *The following propositions hold:*

1. $\cdot \vdash \mathit{intf}_{\mathit{basis}} : \mathsf{Intf}$.
2. *If* $\mathit{pdecs} \vdash \mathit{unitexp} \leadsto \mathit{impl} : \mathit{intf}$ *and* $\vdash \mathit{pdecs}\ \mathsf{ok}$, *then* $\mathit{pdecs} \vdash \mathit{impl} : \mathit{intf}$.
3. *If* $\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topspec} \leadsto \mathit{intf} : \mathsf{Intf}$ *and* $\vdash \mathit{pdecs}\ \mathsf{ok}$, *then* $\mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf}$.
4. *If* $\mathit{udecs} \vdash \mathit{sdecs}; \mathit{tdecs} \leadsto \mathit{intf} : \mathsf{Intf}$ *and* $\vdash \mathit{udecs}, \mathit{sdecs}, \mathit{tdecs}\ \mathsf{ok}$, *then* $\mathit{udecs} \vdash \mathit{decs}$ *and* $\mathit{decs} \vdash \mathit{intf} : \mathsf{Intf}$.
5. *If* $\mathit{decs} \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \leadsto \mathit{impl}$, *then* $\mathit{decs} \vdash \mathit{impl}_0 : \mathit{intf}_0$, $\mathit{decs} \vdash \mathit{intf}_0 \leq \mathit{intf} : \mathsf{Intf}$, *and* $\mathit{decs} \vdash \mathit{impl} : \mathit{intf}$.
6. *If* $\mathit{udecs} \vdash \mathit{topdec} \leadsto \mathit{sbnds} : (\mathit{sdecs}; \mathit{tdecs})$ *and* $\vdash \mathit{udecs}\ \mathsf{ok}$, *then* $\mathit{udecs} \vdash \mathit{decs}$, $\mathit{decs} \vdash \mathit{sbnds} : \mathit{sdecs}$, *and* $\vdash \mathit{udecs}, \mathit{sdecs}, \mathit{tdecs}\ \mathsf{ok}$.
7. *If* $\mathit{udecs} \vdash \mathit{topspec} \leadsto \mathit{sdecs}; \mathit{tdecs}$ *and* $\vdash \mathit{udecs}\ \mathsf{ok}$, *then* $\mathit{udecs} \vdash \mathit{decs}$, $\mathit{decs} \vdash \mathit{sdecs}\ \mathsf{ok}$, *and* $\vdash \mathit{udecs}, \mathit{sdecs}, \mathit{tdecs}\ \mathsf{ok}$.
8. *If* $\mathit{udecs} \vdash \mathit{sigbind} \leadsto \mathit{tdecs}$ *and* $\vdash \mathit{udecs}\ \mathsf{ok}$, *then* $\vdash \mathit{udecs}, \mathit{tdecs}\ \mathsf{ok}$.
9. *If* $\mathit{udecs} \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig}$ *and* $\vdash \mathit{udecs}\ \mathsf{ok}$, *then* $\mathit{udecs} \vdash \mathit{decs}$ *and* $\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig}$.
10. *If* $\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \leadsto \mathit{udecs}, \sigma$ *and* $\vdash \mathit{pdecs}\ \mathsf{ok}$, *then* $\vdash \mathit{udecs}\ \mathsf{ok}$.
11. *If* $\mathit{udecs} \vdash \mathit{funspec} \leadsto \mathit{sdecs}$ *and* $\vdash \mathit{udecs}\ \mathsf{ok}$, *then* $\mathit{udecs} \vdash \mathit{decs}$ *and* $\mathit{decs} \vdash \mathit{sdecs}\ \mathsf{ok}$.
12. *If* $\mathit{udecs} \vdash_{\mathsf{ctx}} \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}$ *and* $\vdash \mathit{udecs}\ \mathsf{ok}$, *then* $\mathit{udecs} \vdash \mathit{decs}$ *and* $\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig}$.
13. *If* $\vdash \mathit{udecs}\ \mathsf{ok}$, *then* $\mathit{udecs} \vdash \mathit{decs}$ *and* $\vdash \mathit{decs}\ \mathsf{ok}$.

prog	::=	*exp* : {}	closed expression of type unit

Table 32: HS linker syntax

Section	*Judgement...*	*Meaning ...*
H.3.1	$\vdash$ *assm* $\leadsto$ *prog*	completion
H.3.2	$\vdash$ *assm* $\leadsto$ *bnds* : *decs*	
H.3.3	$\vdash$ *intf* requires *unitid*	*intf* depends on *unitid*
H.3.4	$\vdash$ *impl* requires *unitid*	*impl* depends on *unitid*
H.3.5	$\vdash$ *prog* ok	*prog* is well-formed

Table 33: HS linker judgements

H.3 Parameters for the Linker

H.3.1 Completion Expression $\boxed{\vdash \mathit{assm} \rightsquigarrow \mathit{prog}}$

$$\frac{\vdash \mathit{assm} \rightsquigarrow \mathit{bnd}_1, \ldots, \mathit{bnd}_n : \mathit{decs} \quad \mathit{var} \notin \mathrm{BV}(\mathit{decs})}{\vdash \mathit{assm} \rightsquigarrow [1 \triangleright \mathit{bnd}_1, \ldots, n \triangleright \mathit{bnd}_n, (n+1) \triangleright \mathit{var} = \{\}].(n+1) : \{\}} \tag{76}$$

H.3.2 Completion Bindings $\boxed{\vdash \mathit{assm} \rightsquigarrow \mathit{bnds} : \mathit{decs}}$

$$\frac{}{\vdash \cdot \rightsquigarrow \cdot : \cdot} \tag{77}$$

$$\frac{\begin{array}{c}\vdash \mathit{assm} \rightsquigarrow \mathit{bnds} : \mathit{decs} \\ \mathit{intf} = \mathit{var} : [\mathit{sdecs}]; \mathit{tdecs}\end{array}}{\begin{array}{l}\vdash \mathit{assm}, \underline{\mathit{basis}} : \mathit{intf} \rightsquigarrow \\ \quad (\mathit{bnds}, \overline{\mathit{basis}} = \mathit{mod}_{\mathit{basis}}) : (\mathit{decs}, \overline{\mathit{basis}} : [\mathit{sdecs}])\end{array}} \tag{78}$$

Rule 78: Since $\vdash \mathit{assm}$ complete, $[\mathit{sdecs}]$ is equivalent to $\mathit{sig}_{\mathit{basis}}$. The structure $\mathit{mod}_{\mathit{basis}}$ must satisfy $\cdot \vdash \mathit{mod}_{\mathit{basis}} : \mathit{sig}_{\mathit{basis}}$; in particular, it must contain at least the following fields:

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

$$\frac{\begin{array}{c}\vdash \mathit{assm} \rightsquigarrow \mathit{bnds} : \mathit{decs} \\ \mathit{intf} = \mathit{var} : [\mathit{sdecs}]; \mathit{tdecs} \\ \mathit{unite} = \mathsf{require}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{mod}\end{array}}{\begin{array}{l}\vdash \mathit{assm}, \underline{\mathit{unitid}} : \mathit{intf} = \mathit{unite} \rightsquigarrow \\ \quad (\mathit{bnds}, \overline{\mathit{unitid}} = \mathit{mod}) : (\mathit{decs}, \overline{\mathit{unitid}} : [\mathit{sdecs}])\end{array}} \tag{79}$$

H.3.3 Interface Dependencies $\boxed{\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}}$

$$\frac{\overline{\mathit{unitid}} \in \mathrm{FV}(\mathit{intf})}{\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}} \tag{80}$$

H.3.4 Implementation Dependencies $\boxed{\vdash \mathit{impl}\ \mathsf{requires}\ \mathit{unitid}}$

$$\frac{\overline{\mathit{unitid}} \in \mathrm{FV}(\mathit{mod})}{\vdash \mathit{mod}\ \mathsf{requires}\ \mathit{unitid}} \tag{81}$$

H.3.5 Well-formed Programs $\boxed{\vdash \mathit{prog}\ \mathsf{ok}}$

$$\frac{\cdot \vdash \mathit{exp} : \{\}}{\vdash \mathit{exp} : \{\}\ \mathsf{ok}} \tag{82}$$

H.3.6 Properties of the HS Parameters for the Linker

Lemma 9 *The following propositions hold:*

1. *If* $\vdash \mathit{assm} \rightsquigarrow \mathit{prog}$ *and* $\cdot \vdash \mathit{assm}\ \mathsf{complete}$, *then* $\vdash \mathit{prog}\ \mathsf{ok}$.
2. *If* $\vdash \mathit{assm} \rightsquigarrow \mathit{bnds} : \mathit{decs}$ *and* $\cdot \vdash \mathit{assm}\ \mathsf{complete}$, *then* $\vdash \mathit{bnds} : \mathit{decs}$.
3. *If* $\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}$ *and* $\mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf}$, *then* $\mathit{unitid} \in \mathrm{dom}(\mathit{pdecs})$.
4. *If* $\vdash \mathit{impl}\ \mathsf{requires}\ \mathit{unitid}$ *and* $\mathit{pdecs} \vdash \mathit{impl} : \mathit{intf}$, *then* $\mathit{unitid} \in \mathrm{dom}(\mathit{pdecs})$.

Section	*Judgement...*	*Meaning ...*
H.4.1	$\vdash prog \Rightarrow res$	*prog* evaluates to *res*

Table 34: HS dynamic semantics

H.4 HS Evaluator

H.4.1 Programs

$\boxed{\vdash prog \Rightarrow res}$

$$\frac{(\cdot, \cdot, [], exp) \hookrightarrow^{\star} (\Delta, \sigma, [], \{\})}{\vdash exp : \{\} \Rightarrow \mathsf{term}} \tag{83}$$

$$\frac{(\cdot, \cdot, [], exp) \hookrightarrow^{\star} (\Delta, \sigma, [], \mathsf{raise}^{\{\}}\ exp_v)}{\vdash exp : \{\} \Rightarrow \mathsf{raise}} \tag{84}$$

I The Definition of Standard ML

I.1 Parameters for the Internal Language

intf	::=	IP, F, G, E, EP	imports, environments, and exports
impl	::=	*unitexp*	basic
		impl : *intf*	coerced to *intf*
Γ	::=	$IE; UE$	import and unit environments

Table 35: MTHM IL syntax

Section	*Judgement...*	*Meaning ...*
I.1.1	$\Gamma \vdash intf : \mathsf{Intf}$	*intf* is well-formed
I.1.2	$\Gamma \vdash impl : intf$	*impl* has interface *intf*
I.1.3	$\Gamma \vdash intf \equiv intf' : \mathsf{Intf}$	interface equivalence
I.1.4	$\Gamma \vdash intf \leq intf' : \mathsf{Intf}$	*intf* is a sub-interface of $intf'$
I.1.5	$pdecs \vdash \Gamma$	Γ declares units in *pdecs*
I.1.6	$\Gamma \vdash \mathsf{open}\ unitid_1 \cdots unitid_n \Rightarrow B$	B declares type names in Γ and top-level identifiers in $unitid_i$
I.1.7	$\vdash \Gamma\ \mathsf{ok}$	Γ is well-formed
MTHM 4.9	$\vdash F\ \mathsf{ok}$	F is well-formed
	$\vdash G\ \mathsf{ok}$	G is well-formed
	$\vdash E\ \mathsf{ok}$	E is well-formed

Table 36: MTHM IL static semantics

Γ	$\in$	$\mathrm{Context} = \mathrm{ImportEnv} \times \mathrm{UnitEnv}$
IE	$\in$	$\mathrm{ImportEnv} = \mathrm{TyNameRef} \overset{\text{fin}}{\rightarrowtail} \mathrm{TyName}$
UE	$\in$	$\mathrm{UnitEnv} = \mathrm{UnitId} \overset{\text{fin}}{\to} \mathrm{IntF}$
$unitid.n$	$\in$	$\mathrm{TyNameRef} = \mathrm{UnitId} \times \mathrm{Nat}$
$intf$	$\in$	$\mathrm{IntF} = \mathrm{Imports} \times \mathrm{FunEnv} \times \mathrm{SigEnv} \times \mathrm{Env} \times \mathrm{Exports}$
IP	$\in$	$\mathrm{Imports} = \mathrm{TyName} \overset{\text{fin}}{\rightarrowtail} \mathrm{TyNameRef}$
EP	$\in$	$\mathrm{Exports} = \mathrm{Nat} \overset{\text{fin}}{\rightarrowtail} \mathrm{TyName}$
$unitid$	$\in$	UnitId (unit identifiers)
n	$\in$	Nat (natural numbers)
B 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	$\in$	$\mathrm{FunEnv} = \mathrm{FunId} \overset{\text{fin}}{\to} \mathrm{FunSig}$
G	$\in$	$\mathrm{SigEnv} = \mathrm{SigId} \overset{\text{fin}}{\to} \mathrm{Sig}$
E or (SE, TE, VE)	$\in$	$\mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv}$
Φ or $(T)(E, (T')E')$	$\in$	$\mathrm{FunSig} = \mathrm{TyNameSet} \times (\mathrm{Env} \times \mathrm{Sig})$
Σ or $(T)E$	$\in$	$\mathrm{Sig} = \mathrm{TyNameSet} \times \mathrm{Env}$
SE	$\in$	$\mathrm{StrEnv} = \mathrm{StrId} \overset{\text{fin}}{\to} \mathrm{Env}$
TE	$\in$	$\mathrm{TyEnv} = \mathrm{TyCon} \overset{\text{fin}}{\to} \mathrm{TyStr}$
VE	$\in$	$\mathrm{ValEnv} = \mathrm{VId} \overset{\text{fin}}{\to} \mathrm{TypeScheme} \times \mathrm{IdStatus}$
(θ, VE)	$\in$	$\mathrm{TyStr} = \mathrm{TypeFcn} \times \mathrm{ValEnv}$
σ or $\forall\alpha^{(k)}.\tau$	$\in$	$\mathrm{TypeScheme} = \bigcup_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type}$
θ or $\Lambda\alpha^{(k)}.\tau$	$\in$	$\mathrm{TypeFcn} = \bigcup_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type}$
τ	$\in$	$\mathrm{Type} = \mathrm{TyVar} \times \mathrm{RowType} \times \mathrm{FunType} \times \mathrm{ConsType}$
$(\alpha_1, \cdots, \alpha_k)$ or $\alpha^{(k)}$	$\in$	TyVar^k
ϱ	$\in$	$\mathrm{RowType} = \mathrm{Lab} \overset{\text{fin}}{\to} \mathrm{Type}$
$\tau \to \tau'$	$\in$	$\mathrm{FunType} = \mathrm{Type} \times \mathrm{Type}$
		$\mathrm{ConsType} = \bigcup_{k \geq 0} \mathrm{ConsType}^{(k)}$
$\tau^{(k)}t$	$\in$	$\mathrm{ConsType}^{(k)} = \mathrm{Type}^k \times \mathrm{TyName}^{(k)}$
$(\tau_1, \cdots, \tau_k)$ or $\tau^{(k)}$	$\in$	Type^k
t	$\in$	TyName (type names)
$funid$	$\in$	FunId (functor identifiers)
$sigid$	$\in$	SigId (signature identifiers)
$strid$	$\in$	StrId (structure identifiers)
$tycon$	$\in$	TyCon (type constructors)
vid	$\in$	VId (value identifiers)
α or $tyvar$	$\in$	TyVar (type variables)
is	$\in$	$\mathrm{IdStatus} = \{\mathtt{c}, \mathtt{e}, \mathtt{v}\}$ (identifier status descriptors)
lab	$\in$	Lab (labels)

Table 37: MTHM static semantic objects

- We denote by $A \overset{\text{fin}}{\rightarrowtail} B$ the set of injective, finite partial functions from A to B. If $f : A \overset{\text{fin}}{\rightarrowtail} B$, then we denote its inverse by $f^{-1} : B \overset{\text{fin}}{\rightarrowtail} A$.
- We assume an injective function $\text{exp} : \text{TyNameSet} \rightarrow \text{Exports}$ that satisfies $\text{rng}(\text{exp}(T)) = T$.
- Interface imports and exports, together with context import environments, support definite references to type names between separately compiled units; in particular, if $\mathit{intf} = \mathit{IP}, F, G, E, \mathit{EP}$ and $\Gamma = \mathit{IE}; \mathit{UE}$, then
  - The sets $\text{dom}(\mathit{IP})$ and $\text{rng}(\mathit{EP})$ bind type names that may occur free in F, G, and E.
  - Imports IP are resolved in context; for example, if $\mathit{IE}(\mathit{unitid}.n) = t'$, then an import $t \mapsto \mathit{unitid}.n$ is resolved by substituting t' for free occurrences of t in intf.
  - Exports EP label the type names bound in intf; for example, if $\mathit{unitid} : \mathit{intf}$ and $\mathit{EP}(n) = t$, then $\mathit{unitid}.n$ is a definite reference to t.
- The notation $\{t/t'\}\mathit{phrase}$ denotes the capture-free substitution of t for free occurrences of t' in phrase where $\mathit{phrase} ::= F \mid G \mid E$.

Table 38: MTHM notation and conventions for the static semantics

$\mathrm{clos} : \mathrm{ImportEnv} \times \mathrm{Basis} \rightharpoonup \mathrm{IntF}$
$\mathrm{clos}(IE, B) = IP, F, G, E, EP$
   if $\mathrm{tynames}\ B \setminus T \subset \mathrm{rng}(IE)$
   where
$$\begin{array}{rcl} B & = & T, F, G, E \\ IP & = & \{t \mapsto IE^{-1}(t)\ ;\ t \in \mathrm{tynames}\ B \setminus T\} \\ EP & = & \exp(T) \end{array}$$

$\mathrm{inst} : \mathrm{Context} \times \mathrm{UnitId} \rightharpoonup \mathrm{Basis}$
$\mathrm{inst}((IE; UE), \mathit{unitid}) = \mathrm{inst}(IE, UE(\mathit{unitid}))$
   if $\mathit{unitid} \in \mathrm{dom}(UE)$ and $(IE, UE(\mathit{unitid})) \in \mathrm{dom}(\mathrm{inst})$

$\mathrm{inst} : \mathrm{ImportEnv} \times \mathrm{IntF} \rightharpoonup \mathrm{Basis}$
$\mathrm{inst}(IE, (IP, F, G, E, EP)) = T, F', G', E'$
   if $\mathrm{dom}(IP) \cap \mathrm{rng}(IE) = \emptyset, \mathrm{rng}(IP) \subset \mathrm{dom}(IE)$
   and $\mathrm{dom}(IP) \cap \mathrm{rng}(EP) = \emptyset$
   where
$$\begin{array}{rcl} T & = & \mathrm{rng}(EP) \\ F' & = & \mathrm{inst}(IE, IP, F) \\ G' & = & \mathrm{inst}(IE, IP, G) \\ E' & = & \mathrm{inst}(IE, IP, E) \end{array}$$

$\mathrm{inst} : \mathrm{ImportEnv} \times \mathrm{Imports} \times \mathit{phrase} \rightharpoonup \mathit{phrase}$
$\mathrm{inst}(IE, IP, \mathit{phrase}) = \{t'_1/t_1\} \cdots \{t'_n/t_n\}\mathit{phrase}$
   if $\mathrm{dom}(IP) \cap \mathrm{rng}(IE) = \emptyset$ and $\mathrm{rng}(IP) \subset \mathrm{dom}(IE)$
   where $\mathrm{dom}(IP) = \{t_1, \ldots, t_n\}$
   and $t'_i = IE(IP(t_i)) \quad \text{for } 1 \leq i \leq n$

Table 39: MTHM $\mathrm{clos}(\cdot)$ and $\mathrm{inst}(\cdot)$

I.1.1 Well-formed Interfaces

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

$$\frac{\begin{array}{c}\vdash \mathit{IE}; \mathit{UE}\ \mathsf{ok}\\ \vdash F\ \mathsf{ok} \quad \vdash G\ \mathsf{ok} \quad \vdash E\ \mathsf{ok} \quad \mathrm{tyvars}\, F \cup \mathrm{tyvars}\, G \cup \mathrm{tyvars}\, E = \emptyset\\ \mathrm{tynames}\, F \cup \mathrm{tynames}\, G \cup \mathrm{tynames}\, E = \mathrm{dom}(\mathit{IP}) \cup \mathrm{rng}(\mathit{EP})\\ \mathrm{dom}(\mathit{IP}) \cap \mathrm{rng}(\mathit{EP}) = \emptyset\\ \mathrm{rng}(\mathit{IP}) \subset \mathrm{dom}(\mathit{IE}) \quad \mathrm{dom}(\mathit{IP}) \cap \mathrm{rng}(\mathit{IE}) = \emptyset\\ \mathrm{rng}(\mathit{EP}) \cap \mathrm{rng}(\mathit{IE}) = \emptyset\end{array}}{\mathit{IE}; \mathit{UE} \vdash (\mathit{IP}, F, G, E, \mathit{EP}) : \mathsf{Intf}} \tag{85}$$

Rule 85: The side-conditions $\mathrm{dom}(\mathit{IP}) \cap \mathrm{rng}(\mathit{IE}) = \emptyset$ and $\mathrm{rng}(\mathit{EP}) \cap \mathrm{rng}(\mathit{IE}) = \emptyset$ can always be satisfied by renaming bound type names.

I.1.2 Well-formed Implementations

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

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \Rightarrow B\\ B \vdash \mathit{topdec} \Rightarrow B' \quad \mathrm{tyvars}\, B' = \emptyset\\ \mathit{intf} = \mathrm{clos}(\mathit{IE}\ \mathrm{of}\ \Gamma, B')\end{array}}{\Gamma \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topdec} : \mathit{intf}} \tag{86}$$

$$\frac{\Gamma \vdash \mathit{impl} : \mathit{intf}' \quad \Gamma \vdash \mathit{intf}' \leq \mathit{intf} : \mathsf{Intf}}{\Gamma \vdash (\mathit{impl} : \mathit{intf}') : \mathit{intf}} \tag{87}$$

I.1.3 Interface Equivalence

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

$$\frac{\Gamma \vdash \mathit{intf} : \mathsf{Intf}}{\Gamma \vdash \mathit{intf} \equiv \mathit{intf} : \mathsf{Intf}} \tag{88}$$

Rule 88: Two interfaces are equivalent if they are identical up to consistent renaming of bound type names.

I.1.4 Sub-interface Relation

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

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{intf} : \mathsf{Intf} \quad \Gamma \vdash \mathit{intf}' : \mathsf{Intf}\\ \mathrm{rng}(\mathit{EP}\ \mathrm{of}\ \mathit{intf}) \cap \mathrm{rng}(\mathit{EP}\ \mathrm{of}\ \mathit{intf}') = \emptyset\\ T, F, G, E = \mathrm{inst}(\mathit{IE}\ \mathrm{of}\ \Gamma, \mathit{intf}) \quad T', F', G', E' = \mathrm{inst}(\mathit{IE}\ \mathrm{of}\ \Gamma, \mathit{intf}')\\ \mathrm{dom}(F) \supset \mathrm{dom}(F') \quad \forall \mathit{funid} \in \mathrm{dom}(F').F'(\mathit{funid}) \geq F(\mathit{funid})\\ \mathrm{dom}(G) \supset \mathrm{dom}(G') \quad \forall \mathit{sigid} \in \mathrm{dom}(G').G'(\mathit{sigid}) \geq G(\mathit{sigid})\\ (T')E' \geq E'' \prec E\end{array}}{\Gamma \vdash \mathit{intf} \leq \mathit{intf}' : \mathsf{Intf}} \tag{89}$$

Rule 89: The side condition $\mathrm{rng}(\mathit{EP}\ \mathrm{of}\ \mathit{intf}) \cap \mathrm{rng}(\mathit{EP}\ \mathrm{of}\ \mathit{intf}') = \emptyset$ can always be satisfied by renaming bound type names.

I.1.5 Context Creation $\boxed{\mathit{pdecs} \vdash \Gamma}$

$$\frac{}{\cdot \vdash \{\},\{\}} \tag{90}$$

$$\frac{\begin{array}{c}\mathit{pdecs} \vdash \mathit{IE}; \mathit{UE} \quad \mathit{unitid} \notin \mathrm{dom}(\mathit{UE}) \quad \mathit{IE}; \mathit{UE} \vdash \mathit{intf} : \mathsf{Intf} \\ \mathit{IE}' = \{t \mapsto \mathit{unitid} \ ; \ t \in \mathrm{rng}(\mathit{EP} \text{ of } \mathit{intf})\} \\ \mathit{UE}' = \{\mathit{unitid} \mapsto \mathit{intf}\}\end{array}}{\mathit{pdecs}, \mathit{unitid} : \mathit{intf} \vdash \mathit{IE} + \mathit{IE}'; \mathit{UE} + \mathit{UE}'} \tag{91}$$

I.1.6 Basis Creation $\boxed{\Gamma \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \Rightarrow B}$

$$\frac{\begin{array}{c}\vdash \Gamma\ \mathsf{ok} \\ B_0 = \mathrm{rng}(\mathit{IE} \text{ of } \Gamma) \text{ in Basis} \\ B_1 = \mathrm{inst}(\Gamma, \mathit{unitid}_1) \quad \cdots \quad B_n = \mathrm{inst}(\Gamma, \mathit{unitid}_n)\end{array}}{\Gamma \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \Rightarrow B_0 + B_1 + \cdots + B_n} \tag{92}$$

I.1.7 Well-formed Contexts $\boxed{\vdash \Gamma\ \mathsf{ok}}$

$$\frac{\begin{array}{l}\forall \mathit{unitid}.n \in \mathrm{dom}(\mathit{IE}). \\ \quad \mathit{unitid} \in \mathrm{dom}(\mathit{UE}) \text{ and} \\ \quad n \in \mathrm{dom}(\mathit{EP} \text{ of } \mathit{UE}(\mathit{unitid})) \\ \forall \mathit{unitid} \mapsto (\mathit{IP}, F, G, E, \mathit{EP}) \in \mathit{UE}. \\ \quad \mathrm{dom}(\mathit{IP}) \cap \mathrm{rng}(\mathit{IE}) = \emptyset, \quad \mathrm{rng}(\mathit{IP}) \subset \mathrm{dom}(\mathit{IE}), \\ \quad \mathrm{dom}(\mathit{IP}) \cap \mathrm{rng}(\mathit{EP}) = \emptyset, \\ \quad \mathrm{tynames}\ F \cup \mathrm{tynames}\ G \cup \mathrm{tynames}\ E = \mathrm{dom}(\mathit{IP}) \cup \mathrm{rng}(\mathit{EP}), \\ \quad \vdash F\ \mathsf{ok}, \quad \vdash G\ \mathsf{ok}, \quad \vdash E\ \mathsf{ok}, \\ \quad \mathrm{tyvars}\ F \cup \mathrm{tyvars}\ G \cup \mathrm{tyvars}\ E = \emptyset, \text{ and} \\ \quad \mathrm{rng}(\mathit{EP}) \cap \mathrm{rng}(\mathit{IE}) = \emptyset,\end{array}}{\vdash \mathit{IE}; \mathit{UE}\ \mathsf{ok}} \tag{93}$$

I.1.8 Properties of the MTHM Parameters for the IL

Lemma 10 *The following propositions hold:*

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

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

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

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

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

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

Section	*Judgement...*	*Meaning ...*
I.2.1	$pdecs \vdash unitexp \rightsquigarrow impl : intf$	unit expressions
I.2.2	$pdecs \vdash \mathsf{open}\ unitid_1 \cdots unitid_n\ \mathsf{in}\ topspec \rightsquigarrow intf$	interface expressions
I.2.3	$B \vdash topspec \Rightarrow B'$	top-level specifications
I.2.4	$B \vdash funspec \Rightarrow F$	functor specifications
I.2.5	$\Gamma \vdash impl_0 : intf_0 \preceq intf \rightsquigarrow impl$	interface ascription

Table 40: MTHM elaborator judgements

The basis interface is defined by $intf_{basis} = \mathrm{clos}(\{\}, B_0)$ where B_0 is the initial static basis defined in MTHM Appendix G.

Table 41: MTHM basis interface

I.2 Parameters for the Elaborator

I.2.1 Unit Expressions

$$\boxed{pdecs \vdash unitexp \rightsquigarrow impl : intf}$$

$$\frac{pdecs \vdash \Gamma \quad \Gamma \vdash unitexp : intf}{pdecs \vdash unitexp \rightsquigarrow unitexp : intf} \tag{94}$$

I.2.2 Interface Expressions

$$\boxed{pdecs \vdash \mathsf{open}\ unitid_1 \cdots unitid_n\ \mathsf{in}\ topspec \rightsquigarrow intf}$$

$$\frac{\begin{array}{c} pdecs \vdash \Gamma \quad \Gamma \vdash \mathsf{open}\ unitid_1 \cdots unitid_n \Rightarrow B \\ B \vdash topspec \Rightarrow B' \quad intf = \mathrm{clos}(IE \text{ of } \Gamma, B') \end{array}}{pdecs \vdash \mathsf{open}\ unitid_1 \cdots unitid_n\ \mathsf{in}\ topspec \rightsquigarrow intf} \tag{95}$$

I.2.3 Top-level Specifications

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

$$\frac{B \vdash spec \Rightarrow E \quad B' = T \text{ of } E, \{\}, \{\}, E \quad \text{tyvars } B' = \emptyset}{B \vdash spec \Rightarrow B'} \tag{96}$$

$$\frac{B \vdash funspec \Rightarrow F \quad B' = T \text{ of } F, F, \{\}, \{\} \quad \text{tyvars } B' = \emptyset}{B \vdash \mathsf{functor}\ funspec \Rightarrow B'} \tag{97}$$

$$\frac{B \vdash sigbind \Rightarrow G \quad B' = T \text{ of } G, \{\}, G, \{\} \quad \text{tyvars } B' = \emptyset}{B \vdash \mathsf{signature}\ sigbind \Rightarrow B'} \tag{98}$$

$$\frac{\begin{array}{c} B \vdash topspec_1 \Rightarrow B_1 \quad B + B_1 \vdash topspec_2 \Rightarrow B_2 \\ \mathrm{dom}(F \text{ of } B_1) \cap \mathrm{dom}(F \text{ of } B_2) = \emptyset \\ \mathrm{dom}(G \text{ of } B_1) \cap \mathrm{dom}(G \text{ of } B_2) = \emptyset \\ \mathrm{dom}(E \text{ of } B_1) \cap \mathrm{dom}(E \text{ of } B_2) = \emptyset \end{array}}{B \vdash topspec_1\ topspec_2 \Rightarrow B_1 + B_2} \tag{99}$$

I.2.4 Functor Specifications

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

$$\frac{\begin{array}{c} B \vdash sigexp \Rightarrow (T)E \quad B \oplus \{strid \mapsto E\} \vdash sigexp' \Rightarrow (T')E' \\ \langle B \vdash funspec \Rightarrow F \quad funid \notin \mathrm{dom}(F) \rangle \end{array}}{\begin{array}{l} B \vdash funid(strid : sigexp) : sigexp'\ \langle \mathsf{and}\ funspec \rangle \Rightarrow \\ \quad \{funid \mapsto (T)(E, (T')E')\}\ \langle +F \rangle \end{array}} \tag{100}$$

I.2.5 Interface Ascription

$$\boxed{\Gamma \vdash impl_0 : intf_0 \preceq intf \rightsquigarrow impl}$$

$$\frac{\Gamma \vdash impl_0 : intf_0 \quad \Gamma \vdash intf_0 \leq intf' : \mathsf{Intf}}{\Gamma \vdash impl_0 : intf_0 \preceq intf' \rightsquigarrow (impl_0 : intf')} \tag{101}$$

I.2.6 Properties of the MTHM Parameters for the Elaborator

Lemma 11 (Well-formed Translation) *The following propositions hold:*

1. $\cdot \vdash \mathit{intf}_{\mathit{basis}} : \mathsf{Intf}$.
2. *If* $\mathit{pdecs} \vdash \mathit{unitexp} \rightsquigarrow \mathit{impl} : \mathit{intf}$, *and* $\vdash \mathit{pdecs}\ \mathsf{ok}$, *then* $\mathit{pdecs} \vdash \mathit{impl} : \mathit{intf}$.
3. *If* $\mathit{pdecs} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topspec} \rightsquigarrow \mathit{intf}$ *and* $\vdash \mathit{pdecs}\ \mathsf{ok}$, *then* $\mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf}$.
4. *If* $B \vdash \mathit{topspec} \Rightarrow B'$ *and* $\vdash B\ \mathsf{ok}$, *then* $\vdash B'\ \mathsf{ok}$ *and* $\mathrm{tyvars}\ B' = \emptyset$.
5. *If* $B \vdash \mathit{funspec} \Rightarrow F$ *and* $\vdash B\ \mathsf{ok}$, *then* $\vdash F\ \mathsf{ok}$.
6. *If* $\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0 \preceq \mathit{intf} \rightsquigarrow \mathit{impl}$, *then* $\Gamma \vdash \mathit{impl}_0 : \mathit{intf}_0$, $\Gamma \vdash \mathit{intf}_0 \leq \mathit{intf} : \mathsf{Intf}$, *and* $\Gamma \vdash \mathit{impl} : \mathit{intf}$.

I.3 Parameters for the Linker

prog	::=	*assm*	complete assembly

Table 42: MTHM linker syntax

Section	*Judgement...*	*Meaning ...*
I.3.1	$\vdash \mathit{assm} \rightsquigarrow \mathit{prog}$	completion
I.3.2	$\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}$	*intf* depends on *unitid*
I.3.3	$\vdash \mathit{impl}\ \mathsf{requires}\ \mathit{unitid}$	*impl* depends on *unitid*
I.3.4	$\vdash \mathit{prog}\ \mathsf{ok}$	*prog* is well-formed

Table 43: MTHM linker judgements

I.3.1 Completion

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

$$\frac{}{\vdash \mathit{assm} \rightsquigarrow \mathit{assm}} \tag{102}$$

I.3.2 Interface Dependencies

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

$$\frac{\exists n.\mathit{unitid}.n \in \mathrm{rng}(\mathit{IP}\ \text{of}\ \mathit{intf})}{\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}} \tag{103}$$

I.3.3 Implementation Dependencies

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

$$\frac{\mathit{unitid} \in \{\mathit{unitid}_1, \ldots, \mathit{unitid}_n\}}{\vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n\ \mathsf{in}\ \mathit{topdec}\ \mathsf{requires}\ \mathit{unitid}} \tag{104}$$

$$\frac{\vdash \mathit{impl}\ \mathsf{requires}\ \mathit{unitid}}{\vdash \mathit{impl} : \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}} \tag{105}$$

$$\frac{\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}}{\vdash \mathit{impl} : \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}} \tag{106}$$

I.3.4 Well-formed Programs

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

$$\frac{\cdot \vdash \mathit{assm}\ \mathsf{complete}}{\vdash \mathit{assm}\ \mathsf{ok}} \tag{107}$$

I.3.5 Properties of the MTHM Parameters for the Linker

Lemma 12 *The following propositions hold:*

1. *If* $\vdash \mathit{assm} \rightsquigarrow \mathit{prog}$ *and* $\cdot \vdash \mathit{assm}\ \mathsf{complete}$*, then* $\vdash \mathit{prog}\ \mathsf{ok}$*.*
2. *If* $\vdash \mathit{intf}\ \mathsf{requires}\ \mathit{unitid}$ *and* $\mathit{pdecs} \vdash \mathit{intf} : \mathsf{Intf}$*, then* $\mathit{unitid} \in \mathrm{dom}(\mathit{pdecs})$*.*
3. *If* $\vdash \mathit{impl}\ \mathsf{requires}\ \mathit{unitid}$ *and* $\mathit{pdecs} \vdash \mathit{impl} : \mathit{intf}$*, then* $\mathit{unitid} \in \mathrm{dom}(\mathit{pdecs})$*.*

I.4 MTHM Evaluator

Section	*Judgement...*	*Meaning ...*
I.4.1	$\vdash \mathit{prog} \Rightarrow \mathit{res}$	*prog* evaluates to *res*
I.4.2	$s, \mathit{UE} \vdash \mathit{assm} \Rightarrow \mathit{UE}'/p, s'$	*assm* evaluates to UE'/p
I.4.3	$s, \mathit{UE} \vdash \mathit{impl} \Rightarrow B/p, s'$	*impl* evaluates to B/p
I.4.4	$\mathit{UE} \vdash \mathsf{open}\ \mathit{unitid}_1 \cdots \mathit{unitid}_n \Rightarrow B$	B binds top-level identifiers in unitid_i

Table 44: MTHM dynamic semantics

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

Table 45: MTHM dynamic semantic objects

- We allow compound metavariables to range over the disjoint union of semantic objects; for example, B/p ranges over $\mathrm{Basis} \cup \mathrm{Pack}$.
- In many cases, the same names are used for static and dynamic semantic objects; for example, E for environments. In this section, such names refer to dynamic semantic objects unless the subscript $(\cdot)_{\mathrm{STAT}}$ is used.

Table 46: MTHM notation and conventions for the dynamic semantics

$\downarrow$: Basis $\times$ Intf $\rightarrow$ Basis
$(F, G, E) \downarrow (IP, F', G', E', EP) = (F \downarrow F', \mathrm{inter}(G'), E \downarrow \mathrm{inter}(E'))$

$\downarrow$: FunEnv $\times$ $\mathrm{FunEnv}_{\mathrm{STAT}}$ $\rightarrow$ FunEnv
$F \downarrow F' = \{\mathit{funid} \mapsto F(\mathit{funid}) \;;\; \mathit{funid} \in \mathrm{dom}(F) \cap \mathrm{dom}(F')\}$

$\downarrow$: Env $\times$ Int $\rightarrow$ Env
defined in MTHM §7.2

inter : $\mathrm{SigEnv}_{\mathrm{STAT}} \rightarrow$ SigEnv
$\mathrm{inter}(G) = \{\mathit{sigid} \mapsto \mathrm{inter}(\Sigma) \;;\; G(\mathit{sigid}) = \Sigma\}$

inter : $\mathrm{Sig}_{\mathrm{STAT}} \rightarrow$ Int
$\mathrm{inter}((T)E) = \mathrm{inter}(E)$

inter : $\mathrm{Env}_{\mathrm{STAT}} \rightarrow$ Int
$\mathrm{inter}(SE, TE, VE) = \mathrm{inter}(SE), \mathrm{inter}(TE), \mathrm{inter}(VE)$

inter : $\mathrm{StrEnv}_{\mathrm{STAT}} \rightarrow$ StrInt
$\mathrm{inter}(SE) = \{\mathit{strid} \mapsto \mathrm{inter}(E) \;;\; SE(\mathit{strid}) = E\}$

inter : $\mathrm{TyEnv}_{\mathrm{STAT}} \rightarrow$ TyInt
$\mathrm{inter}(TE) = \{\mathit{tycon} \mapsto \mathrm{inter}(VE) \;;\; TE(\mathit{tycon}) = (\theta, VE)\}$

inter : $\mathrm{ValEnv}_{\mathrm{STAT}} \rightarrow$ ValInt
$\mathrm{inter}(VE) = \{\mathit{vid} \mapsto \mathit{is} \;;\; VE(\mathit{vid}) = (\sigma, \mathit{is})\}$

Table 47: MTHM $(\cdot \downarrow \cdot)$ and inter$(\cdot)$

I.4.1 Programs

$$\boxed{\vdash \mathit{prog} \Rightarrow \mathit{res}}$$

$$\frac{(\{\},\{\}),\{\} \vdash \mathit{assm} \Rightarrow \mathit{UE}, s}{\vdash \mathit{assm} \Rightarrow \textsf{term}} \tag{108}$$

$$\frac{(\{\},\{\}),\{\} \vdash \mathit{assm} \Rightarrow p, s}{\vdash \mathit{assm} \Rightarrow \textsf{raise}} \tag{109}$$

I.4.2 Assemblies

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

$$\frac{}{s, \mathit{UE} \vdash \cdot \Rightarrow \mathit{UE}, s} \tag{110}$$

$$\frac{\begin{array}{c}\mathrm{dom}(\mathit{mem} \text{ of } s) \cap \mathrm{dom}(\mathit{mem} \text{ of } s_0) = \emptyset \\ (\mathit{ens} \text{ of } s) \cap (\mathit{ens} \text{ of } s_0) = \emptyset \\ s + s_0, \mathit{UE} + \{\mathit{basis} \mapsto B_0\} \vdash \mathit{assm} \Rightarrow \mathit{UE}', s'\end{array}}{s, \mathit{UE} \vdash \mathit{basis} : \mathit{intf}, \mathit{assm} \Rightarrow \mathit{UE}', s'} \tag{111}$$

Rule 111: The initial dynamic basis B_0 is specified in The Definition Appendix D. Its associated state is s_0. The side conditions can always be satisfied by changing addresses and exception names in B_0.

$$\frac{\begin{array}{c}\mathrm{dom}(\mathit{mem} \text{ of } s) \cap \mathrm{dom}(\mathit{mem} \text{ of } s_0) = \emptyset \\ (\mathit{ens} \text{ of } s) \cap (\mathit{ens} \text{ of } s_0) = \emptyset \\ s + s_0, \mathit{UE} + \{\mathit{basis} \mapsto B_0\} \vdash \mathit{assm} \Rightarrow p, s'\end{array}}{s, \mathit{UE} \vdash \mathit{basis} : \mathit{intf}, \mathit{assm} \Rightarrow p, s'} \tag{112}$$

$$\frac{\begin{array}{c}\mathit{unite} = \textsf{require } \mathit{unitid}_1 \cdots \mathit{unitid}_n \textsf{ in } \mathit{impl} \\ s, \mathit{UE} \vdash \mathit{impl} \Rightarrow B, s' \quad s', \mathit{UE} + \{\mathit{unitid} \mapsto B\} \vdash \mathit{assm} \Rightarrow \mathit{UE}', s''\end{array}}{s, \mathit{UE} \vdash \mathit{unitid} : \mathit{intf} = \mathit{unite}, \mathit{assm} \Rightarrow \mathit{UE}', s''} \tag{113}$$

$$\frac{\begin{array}{c}\mathit{unite} = \textsf{require } \mathit{unitid}_1 \cdots \mathit{unitid}_n \textsf{ in } \mathit{impl} \\ s, \mathit{UE} \vdash \mathit{impl} \Rightarrow p, s'\end{array}}{s, \mathit{UE} \vdash \mathit{unitid} : \mathit{intf} = \mathit{unite}, \mathit{assm} \Rightarrow p, s'} \tag{114}$$

$$\frac{\begin{array}{c}\mathit{unite} = \textsf{require } \mathit{unitid}_1 \cdots \mathit{unitid}_n \textsf{ in } \mathit{impl} \\ s, \mathit{UE} \vdash \mathit{impl} \Rightarrow B, s' \quad s', \mathit{UE} + \{\mathit{unitid} \mapsto B\} \vdash \mathit{assm} \Rightarrow p, s''\end{array}}{s, \mathit{UE} \vdash \mathit{unitid} : \mathit{intf} = \mathit{unite}, \mathit{assm} \Rightarrow p, s''} \tag{115}$$

I.4.3 Unit Implementations

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

$$\frac{\begin{array}{c} UE \vdash \mathsf{open}\ unitid_1 \cdots unitid_n \Rightarrow B \\ s, B \vdash topdec \Rightarrow B', s' \end{array}}{s, UE \vdash \mathsf{open}\ unitid_1 \cdots unitid_n\ \mathsf{in}\ topdec \Rightarrow B', s'} \qquad (116)$$

$$\frac{\begin{array}{c} UE \vdash \mathsf{open}\ unitid_1 \cdots unitid_n \Rightarrow B \\ s, B \vdash topdec \Rightarrow p, s' \end{array}}{s, UE \vdash \mathsf{open}\ unitid_1 \cdots unitid_n\ \mathsf{in}\ topdec \Rightarrow p, s'} \qquad (117)$$

$$\frac{s, UE \vdash impl \Rightarrow B, s'}{s, UE \vdash impl : intf \Rightarrow B \downarrow intf, s'} \qquad (118)$$

$$\frac{s, UE \vdash impl \Rightarrow p, s'}{s, UE \vdash impl : intf \Rightarrow p, s'} \qquad (119)$$

I.4.4 Basis Creation

$$\boxed{UE \vdash \mathsf{open}\ unitid_1 \cdots unitid_n \Rightarrow B}$$

$$\frac{B_1 = UE(unitid_1) \quad \cdots \quad B_n = UE(unitid_n)}{UE \vdash \mathsf{open}\ unitid_1 \cdots unitid_n \Rightarrow B_1 + \cdots + B_n} \qquad (120)$$