15-819K: Logic Programming

Final Project Report

A Simple Module System for Twelf

Dan Licata, Rob Simmons, Daniel Lee

December 15, 2006

Abstract

Twelf is an implementation of the LF logical framework and the Elf logic programming language that provides several automated analyses of Elf programs. It is increasingly used in the development and study of large deductive systems, such as programming languages and logics. Such large projects would benefit from language extensions that facilitate extensible code reuse and modular development. Drawing on prior work with module systems for LF and Elf, have designed and prototyped a simple module system for Twelf. The module system supports namespace management and simple forms of parameterization and reuse. It is defined by elaboration into the current Twelf implementation.

1 Introduction

LF [Harper et al., 1993] is a logical framework used to represent deductive systems such as programming languages and logics. Elf [Pfenning, 1989] is a logic programming language that endows LF with an operational semantics based on proof search. Twelf [Pfenning and Schürmann, 1999] is an implementation of LF and Elf that provides various analyses, such as mode, coverage, and termination, of Elf programs. Metatheorems[1] about deductive systems can be represented in Twelf as logic programs that satisfy certain totality analyses. Consequently, Twelf's analyses simultane-

[1] In this report, we will use the words "directive", "analysis", and "assertion" for Twelf directives such as mode, worlds, and totality, whereas we will use the word "metatheorem" more broadly to refer to any proposition about a deductive system.

ously ensure that logic programs will run correctly and verify metatheoretic properties of deductive systems.

Though Twelf is currently used in a wide variety of large research projects (see, for example, Crary [2003], Lee et al. [2007], and Appel [2001]), users often request two additional features:

- **Namespace management:** Current Twelf code is defined in one global namespace, which creates tedious work for developers and prevents modular development (e.g., to integrate two pieces of code, a developer must manually resolve name collisions). The absence of namespace management also impedes the development and distribution of re-usable Twelf libraries.

- **Extensibility:** There is often a need to compose fragments of Twelf signatures into a larger Twelf signature. For example, it is sometimes desirable to reuse an LF type family with two different modes without copying its definition, or to to define fragments of deductive systems and compose them together to form several concrete systems—without copying the common declarations. At present, Twelf provides no support for these circumstances.

Both of these features can be provided by a module system. In prior work, Harper and Pfenning [1998] presented a module system for Elf. In unpublished work, Watkins gave a simplified module language for LF, and Licata, Harper, and Pfenning (LHP) reformulated and implemented a stand-alone prototype of Watkins' design. These module systems support namespace management by permitting code to be structured into hierarchical modules; they also provide conveniences for managing names familiar from ML-style module systems, such as dot notation and `open`. Additionally, these proposals support extensible LF signatures through features familiar from ML, such functors, `include`, and `where type`. However, these proposals account only for LF or early versions of Elf; none are defined for the full Twelf language.

In this project, we have designed a preliminary module language for Twelf. Our module language provides namespace-management and extensibility features similar to those of the Watkins and LHP proposals. The key technical contribution of our design is the formalization of an elaboration relation from the module language into Twelf signatures. We have also produced a prototype implementation of a considerable subset of this elaboration that is designed around making calls into the current Twelf im-

plementation, rather than modifying the Twelf implementation to support the features of the module language.

There is a considerable body of prior work on module systems for logic programming languages and proof assistants. Although there is no standard module system adopted by the various Prologs, a number of Prolog implementations provide module systems. These module systems allow programmers to create modules by defining new predicates or importing definitions from existing modules. Haemmerlé and Fages [2006] make a comparison between the module systems of the SICStus, ECLiPSe, SWIProlog, Ciao Prolog, and XSB Prolog implementations. Miller [1994] proposed a module system for λProlog, which has since found its way into various implementations of the language. In his proposal, Miller gives a semantics for the modular constructs of λProlog in terms of logical connectives already extant in the higher-order theory of hereditary Harrop formulas on which λProlog is based.

There is precedent for module systems in other implementations of logical frameworks/proof assistants. Isabelle [Nipkow et al., 2002] is a theorem prover that can be instantiated over a number of different meta-logics. The Isabelle/HOL theorem prover supports modular reasoning [Kammüller, 2000] with the combined use of *locales* [Kammüller et al., 1999] and dependent types. In Isabelle, locales are a mechanism by which certain assumptions can be given local scope. These are used in conjunction with the Π and Σ types of HOL to create modules. Consequently, this particular technique for using modules in Isabelle/HOL does not necessarily transfer instantiations of Isabelle on meta-logics that lack these dependent types. The Coq proof assistant Coq Development Team [2006] has a module system that supports modules and abstraction via functors. Coq's program extraction utility translates Coq modules, functors, and signatures into the analogous OCaml constructs.

The remainder of this report is organized as follows: In Section 2, we overview the features of our module system. In Section 3, we formally define our module language and its elaboration into Twelf. In Section 4, we briefly describe our implementation of our module system and discuss what would be necessary in order to interface it would the existing Twelf implementation. In Section 5, we discuss future extensions of this work and conclude.

```
nat : type.
z : nat.
s : nat -> nat.
%freeze nat.

tp : type.
tp-unit : tp.
tp-nat : tp.
tp-arrow : tp -> tp.
%freeze tp.

tm : type.
tm-unit : tm.
tm-nat : nat -> tm.
tm-lam :
  tp -> (tm -> tm) -> tm.
tm-app : tm -> tm -> tm.
%freeze tm.
```

```
Nat : %sig
 nat : type.
 z : nat.
 s : nat -> nat.
%end.

Tp : %sig
 tp : type.
 unit : tp.
 nat : tp.
 arrow : tp -> tp.
%end.

Tm : %sig
 tm : type.
 unit : tm.
 nat : Nat.nat -> tm.
 lam :
  Tp.tp -> (tm -> tm) -> tm.
 app : tm -> tm -> tm.
%end.
```

Figure 1: The simply typed lambda calculus with natural numbers, demonstrating how namespace management is done in core Twelf (left) and how it can be done in Twelf with modules (right)

2 Language Overview

This section will describe our module system through a series of examples. In this report, we will use green code to represent "core Twelf," the current Twelf language. We will use red to refer to code written for Twelf extended with a module system.

2.1 Namespace management

In core Twelf, all names are global, and any name can be reused; later instances of names shadow earlier ones. For example, if `tp-unit` and `tm-unit` in Figure 1 were instead both called `unit`, it would be impossible to refer to the `unit` of type `tp` later in the file, the first instance of the identifier would be shadowed by second. A similar representation for the same

```
x : type.

A : %sig
  b = x.
  x : type.
  B : %sig
    x : type.
  %end.

  c = B.x
%end.

d = A.x
e = A.B.x
```

Figure 2: Shadowing in modules.

simply-typed lambda calculus using modules allows for there to be multiple terms referred to by the identifier `unit` which exist in different modules, and therefore in different namespaces. Writing the full path, such as `Nat.nat`, allows us to refer to identifiers in previously defined modules.

Within module signatures, identifiers are required to be unique, with the exception that any number of declarations (such as cases of metatheorems) can be left unnamed by using the existing idiom of naming them "`_`". Because module signatures are a relatively small, local scope rather than the global scope of the Twelf top level, we believe this is a reasonable restriction.

2.2 Scope within modules

Figure 2 demonstrates the way paths and shadowing work within Twelf structures and sub-structures. Variables from previous scopes are available up to the point where they are re-defined, which is what allows the line `b = x` to work. Variables from previously defined modules can be referred to by using the variable's path relative to the current location, which is why the path `B.x` in the definition of `c` does not need to be `A.B.x` like it does in the definition of `e`.

The declaration `%open` allows the programmer to bring all the identifiers from some other module within the current scope. At the top level, `%open` may cause other identifiers to be shadowed. However, inside module sig-

```
Nat : %sig
  nat : type.
  z : nat.
  s : nat -> nat.
%end.

NatPlus : %sig
  %open Nat.

  plus :
   nat -> nat -> nat -> type.
  plus/z : plus z N N.
  plus/s : plus (s N) M (s P)
   <- plus N M P.
%end.
```

```
NAT = %sig
 nat : type.
 z : nat.
 s : nat -> nat.
%end.

NATPLUS = %sig
  %include NAT.

  plus :
   nat -> nat -> nat -> type.
  plus/z : plus z N N.
  plus/s : plus (s N) M (s P)
   <- plus N M P.
%end.

NatPlus : NATPLUS.
```

Figure 3: Parallel examples of creating larger signatures by opening modules (left) and including signatures (right)

natures, an `%open` declaration is subject to the restriction that no names in the opened module collide with other names in the signature. Looking at 2, it would be an error to call `%open B` from within the scope of the module `A`, because the module `A` and the module `B` both have an identifier `x` in them.

2.3 The signature calculus

So far, whenever we have used a `%sig ... %end`, we have immediately taken a module of that signature by writing `M: %sig ... %end`. Furthermore, whenever we have defined a type, such as `nat`, `tm`, or `tp` within a module, we have described all of the objects of those types within the module. These two facts are related; it is our intention that when we create a new a type within a signature, when we take a module of that signature, Twelf will no longer allow more objects of that type to be created.

For cases where we want to define a signature without fully defining the constants of that type, we can declare signature variables. Figure 3 represents how this can be used to write a signature for addition of natural numbers: the example on the left uses `%open` on modules, and the example on the right uses `%include` on signatures. The two examples are somewhat

```
LAM = %sig
  exp : type.
  lam : (exp -> exp) -> exp.
  app : exp -> exp -> exp.
%end.

NAT = %sig
  nat : type.
  z : nat.
  s : nat -> nat
%end.
```

(continued in next column)

```
LamPairs : %sig
  %include LAM.

  pair : exp -> exp -> exp.
%end.

LamNat : %sig
  %include LAM.
  %include NAT.

  n : nat -> exp.
%end.
```

Figure 4: Using `%include` with type variables to create two parallel versions of an untyped lambda calculus, one extended with primitive pairs, and the other extended with primitive natural numbers.

different functionally - in the first example, `Nat.z` and `NatPlus.z` are equivalent constants. In the example at the right, the only way we can access the constant representing zero is `NatPlus.z`. If we were to write `Nat : NAT.` in the example on the right, the `Nat.z` would not be the same as `NatPlus.z`, because taking a module of a signature creates a fresh copy of everything in that signature.

In Figure 4, we see some of the power of the `%include` signature directive for reducing the duplication of code. We can define the basic pieces of the lambda calculus in the signature `LAM`, and then easily extend it in two different ways.

Figure 5 shows a different way of using signatures to reuse code, this time by using `where`. The `where` construct allows components of a module to be instantiated with previously-defined components. In the example in Figure 5, the type family `elet` is instantiated twice, once with the type `nat` and once with the type `exp`. This permits the definition of `LIST` to be reused without textually copying it. The `where` mechanism also permits entire module declarations to be retroactively defined to be an existing module, as shown in the instantiation of `TNat` in Figure 6.

```
exp : type.
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.

nat : type.
z : nat.
s : nat -> nat.

explist : type.
exp_nil : explist.
exp_cons :
  exp -> explist -> explist.

natlist : type.
nat_nil : natlist.
nat_cons :
  nat -> natlist -> natlist.
```

```
exp : type.
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.

nat : type.
z : nat.
s : nat -> nat.

LIST = %sig
  elet : type.
  list : type.
  nil : list.
  cons : elet -> list -> list.
%end.

NatList :
  LIST where elet := nat.
ExpList :
  LIST where elet := exp.
```

Figure 5: List data structures in core Twelf (left) have to be redefined for each type that the programmer needs a list of. Using Twelf with modules (right), the programmer can get many of the benefits of polymorphism on lists by defining the signature `LIST` and then substituting in the correct type with `where`.

```
NAT = %sig
  nat : type.
  z : nat.
  s : nat -> nat.
%end.

TIMES = %sig
  TNat : NAT. %open TNat.

  times :
   nat -> nat -> nat -> type.
  times/z : times z N z.

  %mode times +A +B -C.
  %worlds () (times _ _ _).
  %total T (times T _ _).
%end.

Nat : NAT.
```

(continued in next column)

```
Plus : %sig
  %open Nat.

  plus :
   nat -> nat -> nat -> type.
  plus/z : plus z N N.
  plus/s : plus (s N) M (s P)
   <- plus N M P.

  %mode plus +A +B -C.
  %worlds () (plus _ _ _).
  %total T (plus T _ _).
%end.

Times : %sig
  %include TIMES
  %open Plus.

  times/s : times (s N) M Q
   <- times N M P
   <- plus M P Q.
%end where TNat := Nat.
```

Figure 6: Demonstrates the delayed running of Twelf directives. The totality assertion for `times` is false based on the information in the signature `TIMES` (left), but additional information given when the signature is extended allows the totality assertion to hold when it is checked (right).

2.4 Twelf directives

The primary difference between a module system for LF or Elf and a module system for Twelf is that a module system for Twelf must account for the large number of directives which can have effects on the behavior of Twelf. The module system design must decide at what point these directives are run.

Our module system design takes the following approach. A signature can include un-enforced or un-verified Twelf directives, but it must contain only valid LF terms. When a top-level module of a signature is created, the signature is split into two parts. The first is a pure LF signature, along with certain Twelf directives that can be run eagerly to improve error messages. Currently the only such directive that we implement is `%name`, though others could be added. The pure LF part only needs a check to ensure that it is not extending any type families which Twelf has frozen. After this has occurred, the remaining Twelf directives are run. For example, this design permits a module signature to contain a totality assertion for a type family before before all the constants implementing that family have been specified. It is only when we make a module out of the signature that the totality of the type family is checked. This is demonstrated in Figure 6.

The full list of directives that are split off and not run until the signature is made into a module is `%query`, `%fquery`, `%querytabled`, `%mode`, `%unique`, `%covers`, `%total`, `%terminates`, `%reduces`, `%theorem`, `%prove`, `%establish`, `%assert`, `%worlds`, `%deterministic`, and `%clause`.

Additionally, we disallow some declarations from appearing in module signatures at all. Essentially, these are directives which would not behave well under the signature-splitting action. `%define` and `%solve` introduce constants that later LF objects might depend on, and the meaning of `%freeze` and `%thaw` declarations would change if they were moved—besides, within a signature the freezing that happens automatically at module boundaries should be sufficient. Finally, the `%use` declaration introducing a constraint domain would not have any obvious meaning within a signature.

3 Language Definition

3.1 Elaboration Target: Twelf

Our interface to Twelf consists of the following abstract syntax:

Name	x	::=	id $\mid \_$
Expression	e	::=	$x \mid \mathsf{type} \mid \Pi\, x{:}e.\, e' \mid e\, e' \mid \lambda\, x{:}e.\, e' \mid e : e'$
Signature	Σ	::=	$\cdot \mid \Sigma, d$
Declaration	d	::=	$x : e \mid x : e =_a e' \mid \mathsf{block}\, x = bs \mid x : \%\alpha$
Abbreviation Flag	a	::=	$\cdot \mid \%\mathsf{abbrev}$
Block Specification	bs	::=	$\mathsf{some}\ (\overline{x_i : e_i})\ \mathsf{block}\ (\overline{x_j : e_j})$
Directives	$\%\alpha$	::=	...

Expressions e range over all LF objects, families, and kinds. Twelf signatures Σ consist of constant declarations, constant definitions and abbreviations (tagged with $\%$abbrev), block declarations, and Twelf directives. Our elaboration judgements assume that Twelf directives are named in Σ. We elide the enumeration of all Twelf directives $\%\alpha$.

We assume Twelf provides the following judgements:

- $\boxed{\Sigma \vdash e\ \mathsf{class}}$ $\boxed{\Sigma \vdash e \equiv e'\ \mathsf{class}}$ $\boxed{\Sigma \vdash x\ \mathsf{famconst}}$ The first two judgements define formation and equality of classifiers (types and kinds). This judgement depends only on LF notions, and, for example, does not take account of the Twelf notion of freezing. The third identifies family-level constants.

- $\boxed{\Sigma \vdash e : e'}$ $\boxed{\Sigma \vdash e \equiv e' : e''}$ These judgements define formation and equality of expressions at classifiers (objects at types and families at kinds).

- $\boxed{\Sigma \vdash bs\ \mathsf{blockspec}}$ $\boxed{\Sigma \vdash bs \equiv bs'\ \mathsf{blockspec}}$ These judgements define formation and equality of block specifications, where equality of block specifications lifts equality of the embedded expressions.

- $\boxed{\Sigma \vdash \%\alpha\ \mathsf{dir}}$ $\boxed{\Sigma \vdash \%\alpha \equiv \%\alpha'\ \mathsf{dir}}$ These judgements define formation and equality of directives. Formation of directives checks that all identifiers and terms are well-formed; equality lifts equality of the embedded expressions.

- $\boxed{\Sigma \vdash \%\alpha\ \mathsf{succeeds}}$ Check that the directive succeeds.

- $\boxed{\Sigma \vdash \Sigma'\ \mathsf{sig}}$ Check that the Twelf signature Σ' is well-formed.

3.2 Syntax of Module Language

The abstract syntax of our module and signature extension is defined by the following grammar:

Label	l	::=	id
Label Or Underscore	L	::=	l | _
Expression	e	::=	x | type | Π x:e. e′ | e e′ | λ x:e. e′ | e : e′ | p
Path	p	::=	l | l.p
Shared Declaration	d	::=	L : e | L : e =$_a$ e′ | block l = bs | L : %α | l : S | l : S = p | open p
Mod. Sig. Decl.	D	::=	· | d. D | include S. D | block l. D | block l = p. D | L : %α = p. D
Module Signature	S	::=	sig D end | S where N | l
Mod. Sig. Inst.	N	::=	p := e | p := bs | p := p′
Block Specification	bs	::=	some ($\overline{x_i : e_i}$) block ($\overline{x_j : e_j}$)
Top-Level Signature	Σ	::=	· | Σ,d | Σ,l=S

The language of LF expressions is extended with paths p, which are non-empty sequences of identifiers. Some declarations d are shared between module signatures and top-level signatures: LF constant declarations and definitions, block declarations, Twelf directives, structure declarations and definitions, and opened modules. Module signature declarations D contain several additional declarations which permit one signature to be included into another, undefined blocks to be declared (for the purpose of later instantiations), and blocks and Twelf directives to be defined to be paths (which must describe equivalent blocks and Twelf directives). Module signatures S consist of literal sequences of declarations, instantiations, and labels. Module signature instantiations N permit a component of a module signature to be instantiated with expressions, block specifications, or other paths (describing blocks, directives, or structures). Top-level signatures Σ contain shared declarations and named signatures.

3.3 Elaboration Data Structures

The elaboration of the above module language into Twelf requires several auxiliary data structures and operations on them. The syntax of these data structures is defined by the following grammar:

Elaborated Module Sig.	D	::=	$\cdot \mid \mathsf{l} \rhd \mathsf{x} : \mathsf{e}.\ \mathsf{D} \mid \mathsf{l} \rhd \mathsf{x} : \mathsf{e} =_{\mathsf{a}} \mathsf{e}'.\ \mathsf{D}$ $\mid \mathsf{block}\ \mathsf{l} \rhd \mathsf{x}.\ \mathsf{D} \mid \mathsf{block}\ \mathsf{l} \rhd \mathsf{x} = \mathsf{bs}.\ \mathsf{D}$ $\mid \mathsf{l} \rhd \%\alpha.\ \mathsf{D}$
Signature Name Context	ψ	::=	$\cdot \mid \psi, \mathsf{l} = (\mathsf{D}, \delta)$
Module Signature Directory	δ	::=	$\cdot \mid \delta, \mathsf{L} \rhd \mathsf{x} \mid \delta, \mathsf{L} \rhd \mathsf{l} \mid \delta, \mathsf{l} \rhd \delta$
Mod. Sig. Directory Result	r	::=	$\mathsf{l} \mid \mathsf{x} \mid \delta$
Signature Directory	σ	::=	$\cdot \mid \sigma, \mathsf{L} \rhd \mathsf{x} \mid \sigma, \mathsf{l} \rhd \sigma'$

Elaborated module signatures D are similar to Twelf signatures Σ, but they differ in several ways. First, in D the name x in each declaration is considered to be bound in the remainder of the module signature. Second, the components of D are identified by labels l—the names x cannot be used to identify components because they are subject to α-conversion. Finally, D permits uninstantiated block declarations that can be defined by future instantiations. Elaborated module signatures are flat, rather than hierarchical, which obviates the need for paths in expressions, block specifications, and directives.

A signature name context ψ maps a label to an elaborated module signature and a directory. A directory captures the structure that a source language had before elaboration by mapping source-language paths to components of an elaborated module signature D. Specifically, δ maps a label to a label indexing into D, to another directory representing a substructure, or to names in the ambient context. The ability to map paths to names is used to account for definitions that arise during elaboration, as we will see below. We use a subsyntax σ of δ for directories whose leaves all map to names; such a directories will be used to map paths into contexts Σ.

Elaboration relies on several auxiliary operations on these data structures whose behavior is simple to specify but whose definitions are sometimes complex. Consequently, we first specify these auxiliary operations in the remainder and present the main elaboration rules. Section 3.5 contains the definitions of these judgements.

The auxiliary judgements have the following forms:

- $\boxed{\mathsf{L} \# \delta}$ $\boxed{\delta \# \delta'}$ The first judgement holds when l is underscore or when L is an l is apart from the domain of δ. The second holds when all labels in the domain of the first lie apart from the domain of the second.
- $\boxed{\delta(\mathsf{l}) = \mathsf{r}}$ $\boxed{\delta(\mathsf{l}.\mathsf{p}) = \mathsf{r}}$ Look up a label or a path in a directory.
- $\boxed{[\mathsf{x} \leftarrow \mathsf{l}]\delta = \delta'}$ and $\boxed{[\mathsf{l} \leftarrow \mathsf{x}]\delta = \delta'}$ Replace identifiers in the range of δ. The second judgement does not replace labels in paths; for example,

$[\mathsf{l} \leftarrow \mathsf{x}]\mathsf{l} \rhd \mathsf{l} = \mathsf{l} \rhd \mathsf{x}$.

- $\boxed{\mathsf{D}; \delta \rightrightarrows_s \Sigma_\rho \,;\, \Sigma_\omega \,;\, \sigma}$ where $s = m \,|\, t$ is a tag distinguishing being called from a module signature (m) from being called from a top-level signature (t). This judgement splits D into two Twelf signatures, Σ_ρ consisting of LF and Twelf declarations that are known to succeed (assuming that the input D is well-formed), and Σ_ω consisting of suspended Twelf directives that have not yet been run. The judgement assumes that δ contains all and only paths referencing each component of D. It also returns a σ where references into D are replaced with references into $\Sigma_\rho, \Sigma_\omega$.

- $\boxed{(\Sigma_\rho \,;\, \Sigma_\omega \sim \mathsf{D}_1)@(\mathsf{D}_2, \delta_2) = (\mathsf{D}_{12}, \delta_2')}$ Assuming $\Sigma_\rho, \Sigma_\omega$ contains one declaration for each component of D_1, this judgement appends D_2 into the tail of D_1, capturing the names in $\Sigma_\rho, \Sigma_\omega$—i.e., the judgement introduces binders for each component of D_1 binding the names in the signatures. The judgement also replaces references in δ_2 to names in $\Sigma_\rho, \Sigma_\omega$ with references to labels in D_{12}.

- $\boxed{\mathsf{D}|\delta = \mathsf{D}'; \Sigma_\rho; \Sigma_\omega}$ Assuming that δ references a contiguous fragment of D, return that fragment as D', along with contexts representing the bindings in D whose scope D' is in.

- $\boxed{\sigma \,/\, \delta = \beta}$ where $\beta ::= \cdot \,|\, \beta, \mathsf{x}/\mathsf{l}$ When the paths in σ are identical to the paths in δ that lead to labels, this judgement creates an association between the name x at the end of a path in σ and the label l at the end of the same path in δ.

- $\boxed{(\mathsf{D}; \delta)[\beta] = (\mathsf{D}'; \delta')}$ When the labels in β name a subsequence of D, this judgement replaces the components of D labeled in β with the associated name, also rerouting references in δ to those labels to the name.

- $\boxed{\mathsf{freezes}(\Sigma) = \Sigma'}$ Σ' consists of a freeze directive for each type family declared in Σ.

We also use the notation $\%\alpha$ run for Twelf directives that are run while elaborating module signatures and $\%\alpha$ wait for directives that may appear in signatures but are not run until a module is created at the top level (recall the classification in Section 2.4).

3.4 Main Elaboration Judgements

The main elaboration judgements have the following forms:

- $\boxed{\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}}$ $\boxed{\sigma \vdash \mathsf{bs} \rightsquigarrow \mathsf{bs}}$ $\boxed{\sigma \vdash \%\alpha \rightsquigarrow \%\alpha}$ This judgement replaces paths with names.
- $\boxed{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash_\psi \mathsf{D} \rightsquigarrow \mathsf{D}\,;\,\delta}$ $\boxed{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash_\psi \mathsf{S} \rightsquigarrow \mathsf{D}\,;\,\delta}$ These judgements elaborate a module signature or module declarations to an elaborated module signature and a directory of paths into it.
- $\boxed{\Sigma_\rho;\Sigma_\omega \vdash \sigma \;\leq\; (\mathsf{D}\,;\,\delta)\,|\,\sigma'}$ This judgement checks that σ is a subsignature of δ, in the sense that the paths in σ into $\Sigma_\rho, \Sigma_\omega$ supply equivalent components to all paths in δ into D or $\Sigma_\rho, \Sigma_\omega$. It yields σ', which is exactly those paths in σ used to satisfy components of δ.
- $\boxed{\Sigma_\rho;\Sigma_\omega;\sigma \vdash \mathsf{N} : (\mathsf{D},\delta) \Rightarrow (\mathsf{D}',\delta')}$ This judgement applies the instantiation N to D and δ, yielding D' and δ'.
- $\boxed{\Sigma \rightsquigarrow \S\,;\,\sigma\,;\,\psi}$ This judgement elaborates a top-level signature.

$\boxed{\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}}$

$$\frac{\sigma(\mathsf{p}) = \mathsf{x}}{\sigma \vdash \mathsf{p} \rightsquigarrow \mathsf{x}} \qquad \frac{}{\sigma \vdash \mathsf{x} \rightsquigarrow \mathsf{x}} \qquad \frac{}{\sigma \vdash \mathsf{type} \rightsquigarrow \mathsf{type}}$$

$$\frac{\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e} \quad \mathsf{x} \# \sigma \quad \sigma \vdash \mathsf{e}' \rightsquigarrow \mathsf{e}'}{\sigma \vdash \Pi\,\mathsf{x}{:}\mathsf{e}.\,\mathsf{e}' \rightsquigarrow \Pi\,\mathsf{x}{:}\mathsf{e}.\,\mathsf{e}'} \qquad \frac{\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e} \quad \mathsf{x} \# \sigma \quad \sigma \vdash \mathsf{e}' \rightsquigarrow \mathsf{e}'}{\sigma \vdash \lambda\,\mathsf{x}{:}\mathsf{e}.\,\mathsf{e}' \rightsquigarrow \lambda\,\mathsf{x}{:}\mathsf{e}.\,\mathsf{e}'}$$

$$\frac{\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}' \quad \sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}'}{\sigma \vdash \mathsf{e}\,\mathsf{e}' \rightsquigarrow \mathsf{e}\,\mathsf{e}'} \qquad \frac{\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e} \quad \sigma \vdash \mathsf{e}' \rightsquigarrow \mathsf{e}'}{\sigma \vdash \mathsf{e} : \mathsf{e}' \rightsquigarrow \mathsf{e} : \mathsf{e}'}$$

$\boxed{\sigma \vdash \mathsf{bs} \rightsquigarrow \mathsf{bs}}$

$$\frac{\mathsf{x_i}, \mathsf{x_j} \# \sigma \quad \sigma \vdash \mathsf{e_i} \rightsquigarrow \mathsf{e_i} \quad \sigma \vdash \mathsf{e_j} \rightsquigarrow \mathsf{e_j}}{\sigma \vdash \mathsf{some}\ (\overline{\mathsf{x_i} : \mathsf{e_i}})\ \mathsf{block}\ (\overline{\mathsf{x_j} : \mathsf{e_j}}) \rightsquigarrow \mathsf{some}\ (\overline{\mathsf{x_i} : \mathsf{e_i}})\ \mathsf{block}\ (\overline{\mathsf{x_j} : \mathsf{e_j}})}$$

$\boxed{\sigma \vdash \%\alpha \rightsquigarrow \%\alpha}$ We elide the definition of this judgement.

$\boxed{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash_\psi \mathsf{D} \rightsquigarrow \mathsf{D}\,;\,\delta}$

ψ is invariant throughout a derivation, so we elide it from the rules.

$$\frac{\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e} \quad \Sigma_\rho \vdash \mathsf{e}\,\mathsf{class} \quad \mathsf{x}\,\mathsf{fresh} \quad \Sigma_\rho, \mathsf{x}:\mathsf{e}\,;\Sigma_\omega\,;\sigma, \mathsf{L} \rhd \mathsf{x} \vdash \mathsf{D} \rightsquigarrow \mathsf{D}\,;\,\delta \quad \mathsf{L} \# \delta \quad \mathsf{l}'\,\mathsf{fresh}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash (\mathsf{L}:\mathsf{e}.\,\mathsf{D}) \rightsquigarrow (\mathsf{l}' \rhd \mathsf{x}:\mathsf{e}.\,\mathsf{D})\,;\,(\mathsf{L} \rhd \mathsf{l}', [\mathsf{l}' \leftarrow \mathsf{x}]\delta)}$$

We clarify two subtleties of this rule. First, the directory δ returned by the inductive call may contain references to the variable x added to the signature; the substitution in the conclusion of the rule reroutes these references to the appropriate component of the return value. Second, the check $\mathsf{L} \# \delta$ ensures that all identifier labels in the signature are unique (underscore can appear more than once).

$$\frac{\begin{array}{c}\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e} \\ \sigma \vdash \mathsf{e}' \rightsquigarrow \mathsf{e}' \\ \Sigma_\rho \vdash \mathsf{e}' : \mathsf{e} \\ \mathsf{x}\,\mathsf{fresh} \\ \Sigma_\rho, \mathsf{x}:\mathsf{e} =_\mathsf{a} \mathsf{e}'\,;\Sigma_\omega\,;\sigma, \mathsf{L} \rhd \mathsf{x} \vdash \mathsf{D} \rightsquigarrow \mathsf{D}\,;\,\delta \\ \mathsf{L} \# \delta \\ \mathsf{l}'\,\mathsf{fresh}\end{array}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash (\mathsf{L}:\mathsf{e} =_\mathsf{a} \mathsf{e}'.\,\mathsf{D}) \rightsquigarrow (\mathsf{l}' \rhd \mathsf{x}:\mathsf{e} =_\mathsf{a} \mathsf{e}'.\,\mathsf{D})\,;\,(\mathsf{L} \rhd \mathsf{l}', [\mathsf{l}' \leftarrow \mathsf{x}]\delta)}$$

$$\frac{\begin{array}{c}\sigma \vdash \mathsf{bs} \rightsquigarrow \mathsf{bs} \\ \Sigma_\rho \vdash \mathsf{bs}\,\mathsf{blockspec} \\ \mathsf{x}\,\mathsf{fresh} \\ \Sigma_\rho, \mathsf{block}\,\mathsf{x} = \mathsf{bs}\,;\Sigma_\omega\,;\sigma, \mathsf{l} \rhd \mathsf{x} \vdash \mathsf{D} \rightsquigarrow \mathsf{D}\,;\,\delta \\ \mathsf{l} \# \delta \\ \mathsf{l}'\,\mathsf{fresh}\end{array}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash (\mathsf{block}\,\mathsf{l} = \mathsf{bs}.\,\mathsf{D}) \rightsquigarrow (\mathsf{block}\,\mathsf{l}' \rhd \mathsf{x} = \mathsf{bs}.\,\mathsf{D})\,;\,(\mathsf{l} \rhd \mathsf{l}', [\mathsf{l}' \leftarrow \mathsf{x}]\delta)}$$

$$\frac{\begin{array}{c}\sigma \vdash \%\alpha \rightsquigarrow \%\alpha \\ \%\alpha\,\mathsf{run} \\ \Sigma_\rho \vdash \%\alpha\,\mathsf{succeeds} \\ \mathsf{x}\,\mathsf{fresh} \\ \Sigma_\rho, \mathsf{x}:\%\alpha\,;\Sigma_\omega\,;\sigma, \mathsf{L} \rhd \mathsf{x} \vdash \mathsf{D} \rightsquigarrow \mathsf{D}\,;\,\delta \\ \mathsf{l}'\,\mathsf{fresh}\end{array}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash (\mathsf{L}:\%\alpha.\,\mathsf{D}) \rightsquigarrow (\mathsf{l}' \rhd \%\alpha.\,\mathsf{D})\,;\,(\mathsf{L} \rhd \mathsf{l}', [\mathsf{x} \leftarrow \mathsf{l}']\delta)}$$

$$\frac{\sigma \vdash \%\alpha \rightsquigarrow \%\alpha \quad \%\alpha\,\mathsf{wait} \quad \mathsf{x}\,\mathsf{fresh} \quad \Sigma_\rho\,;\Sigma_\omega,\mathsf{x}:\%\alpha\,;\sigma,\mathsf{L}\rhd\mathsf{x}\vdash \mathsf{D}\rightsquigarrow\mathsf{D}\,;\,\delta \quad \mathsf{l}'\,\mathsf{fresh}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma\vdash(\mathsf{L}:\%\alpha.\ \mathsf{D})\rightsquigarrow(\mathsf{l}'\rhd\%\alpha.\ \mathsf{D})\,;\,(\mathsf{L}\rhd\mathsf{l}',[\mathsf{x}\leftarrow\mathsf{l}']\delta)}$$

$$\frac{\begin{array}{c}\Sigma_\rho\,;\Sigma_\omega\,;\sigma\vdash \mathsf{S}_1\rightsquigarrow\mathsf{D}_1\,;\,\delta_1\\ \mathsf{D}_1;\delta_1\ \rightrightarrows_m\ \Sigma_{\rho 1}\,;\,\Sigma_{\omega 1}\,;\,\sigma_1\\ \Sigma_\rho,\Sigma_{\rho 1}\,;\Sigma_\omega,\Sigma_{\omega 1}\,;\sigma,\mathsf{l}\rhd\sigma_1\vdash\mathsf{D}_2\rightsquigarrow\mathsf{D}_2\,;\,\delta_2\\ \mathsf{l}\#\delta_2\\ (\Sigma_{\rho 1}\,;\,\Sigma_{\omega 1}\ \sim\ \mathsf{D}_1)@(\mathsf{D}_2,\delta_2)=(\mathsf{D}_{12},\delta_2')\end{array}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma\vdash(\mathsf{l}:\mathsf{S}_1.\ \mathsf{D}_2)\rightsquigarrow\mathsf{D}_{12}\,;\,(\mathsf{l}\rhd\delta_1,\delta_2')}$$

The following rule exemplifies why we permit references into the ambient context in δ: this mechanism is used to resolve the defined substructure. Note that D_2 from the inductive call is returned directly.

$$\frac{\begin{array}{c}\Sigma_\rho\,;\Sigma_\omega\,;\sigma\vdash \mathsf{S}_1\rightsquigarrow\mathsf{D}_1\,;\,\delta_1\\ \sigma(\mathsf{p})=\sigma_1\\ \Sigma_\rho;\Sigma_\omega\vdash\sigma_1\ \leq\ (\mathsf{D}_1\,;\,\delta_1)\,|\,\sigma_1'\\ \Sigma_\rho\,;\Sigma_\omega\,;\sigma,\mathsf{l}\rhd\sigma_1'\vdash\mathsf{D}_2\rightsquigarrow\mathsf{D}_2\,;\,\delta_2\\ \mathsf{l}\#\delta_2\end{array}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma\vdash(\mathsf{l}:\mathsf{S}_1=\mathsf{p}_1.\ \mathsf{D}_2)\rightsquigarrow\mathsf{D}_2\,;\,(\mathsf{l}\rhd\sigma_1',\delta_2)}$$

$$\frac{\sigma(\mathsf{p})=\sigma_1 \quad \Sigma_\rho\,;\Sigma_\omega\,;\sigma,\sigma_1\vdash\mathsf{D}\rightsquigarrow\mathsf{D}\,;\,\delta \quad \sigma_1\#\delta}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma\vdash(\mathsf{open}\,\mathsf{p}.\ \mathsf{D})\rightsquigarrow\mathsf{D}\,;\,(\sigma_1,\delta)}$$

$$\frac{\begin{array}{c}\Sigma_\rho\,;\Sigma_\omega\,;\sigma\vdash \mathsf{S}_1\rightsquigarrow\mathsf{D}_1\,;\,\delta_1\\ \mathsf{D}_1;\delta_1\ \rightrightarrows_m\ \Sigma_{\rho 1}\,;\,\Sigma_{\omega 1}\,;\,\sigma_1\\ \Sigma_\rho,\Sigma_{\rho 1}\,;\Sigma_\omega,\Sigma_{\omega 1}\,;\sigma,\sigma_1\vdash\mathsf{D}_2\rightsquigarrow\mathsf{D}_2\,;\,\delta_2\\ \delta_1\#\delta_2\\ (\Sigma_{\rho 1}\,;\,\Sigma_{\omega 1}\ \sim\ \mathsf{D}_1)@(\mathsf{D}_2,\delta_2)=(\mathsf{D}_{12},\delta_2')\end{array}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma\vdash(\mathsf{include}\,\mathsf{S}_1.\ \mathsf{D}_2)\rightsquigarrow\mathsf{D}_{12}\,;\,(\delta_1,\delta_2')}$$

$$\frac{\mathsf{x}\,\mathsf{fresh} \quad \Sigma_\rho,\mathsf{block}\,\mathsf{x}=\mathsf{some}\,(\cdot)\,\mathsf{block}\,(\cdot)\,;\Sigma_\omega\,;\sigma,\mathsf{l}\rhd\mathsf{x}\vdash\mathsf{D}\rightsquigarrow\mathsf{D}\,;\,\delta \quad \mathsf{l}\#\delta \quad \mathsf{l}'\,\mathsf{fresh}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma\vdash(\mathsf{block}\,\mathsf{l}.\ \mathsf{D})\rightsquigarrow(\mathsf{block}\,\mathsf{l}'\rhd\mathsf{x}.\ \mathsf{D})\,;\,(\mathsf{l}\rhd\mathsf{l}',[\mathsf{l}'\leftarrow\mathsf{x}]\delta)}$$

$$\frac{\sigma(\mathsf{p}) = \mathsf{x} \quad \mathsf{block}\,\mathsf{x} = \mathsf{bs}\ \text{in}\ \Sigma_\rho \quad \Sigma_\rho\,;\Sigma_\omega\,;\sigma,\mathsf{I}\rhd\mathsf{x} \vdash \mathsf{D} \rightsquigarrow \mathsf{D}\,;\,\delta}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash (\mathsf{block}\,\mathsf{I} = \mathsf{p}.\,\mathsf{D}) \rightsquigarrow \mathsf{D}\,;\,(\mathsf{I}\rhd\mathsf{x},\delta)}$$

$$\frac{\begin{array}{c}\sigma \vdash \%\alpha' \rightsquigarrow \%\alpha' \\ \sigma(\mathsf{p}) = \mathsf{x} \\ \mathsf{x} : \%\alpha\ \text{in}\ \Sigma_\rho, \Sigma_\omega \\ \Sigma_\rho \vdash \%\alpha \equiv \%\alpha'\ \mathsf{dir} \\ \Sigma_\rho\,;\Sigma_\omega\,;\sigma,\mathsf{I}\rhd\mathsf{x} \vdash \mathsf{D} \rightsquigarrow \mathsf{D}\,;\,\delta\end{array}}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash (\mathsf{I} : \%\alpha' = \mathsf{p}.\,\mathsf{D}) \rightsquigarrow \mathsf{D}\,;\,(\mathsf{I}\rhd\mathsf{x},\delta)}$$

$\boxed{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash_\psi \mathsf{S} \rightsquigarrow \mathsf{D}\,;\,\delta}$

$$\frac{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash_\psi \mathsf{D} \rightsquigarrow \mathsf{D}\,;\,\delta}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash_\psi \mathsf{sig}\,\mathsf{D}\,\mathsf{end} \rightsquigarrow \mathsf{D}\,;\,\delta}$$

$$\frac{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash_\psi \mathsf{S} \rightsquigarrow \mathsf{D}\,;\,\delta \quad \Sigma_\rho;\Sigma_\omega;\sigma \vdash \mathsf{N} : (\mathsf{D},\delta) \Rightarrow (\mathsf{D}',\delta')}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash_\psi \mathsf{S}\,\mathsf{where}\,\mathsf{N} \rightsquigarrow \mathsf{D}'\,;\,\delta'}$$

$$\frac{\mathsf{I} = (\mathsf{D},\delta)\ \text{in}\ \psi}{\Sigma_\rho\,;\Sigma_\omega\,;\sigma \vdash_\psi \mathsf{I} \rightsquigarrow \mathsf{D}\,;\,\delta}$$

$\boxed{\Sigma_\rho;\Sigma_\omega \vdash \sigma \le (\mathsf{D}\,;\delta) \,|\, \mathsf{D}';\sigma'}$ Inductively, this judgement returns leftover declarations D not matched by paths in σ as D'. In the rules, we elide Σ_ρ and Σ_ω because they are invariant. The rules assume that the labels in the range of δ are exactly the components of D and that a depth-first traversal of δ finds these labels in the same order they occur in D. In the definitions of other judgements, we write $\Sigma_\rho;\Sigma_\omega \vdash \sigma \le (\mathsf{D}\,;\,\delta) \,|\, \sigma'$ to mean $\sigma \le (\mathsf{D}\,;\,\delta) \,|\, \cdot;\sigma'$, which is the condition under which an overall query succeeds.

Leftovers:

$$\frac{}{\cdot \le (\mathsf{D}\,;\,\delta) \,|\, \mathsf{D};\cdot}$$

Dropped fields:

$$\frac{}{\sigma \le (\cdot\,;\,\cdot) \,|\, \cdot;\cdot}$$

The following rule permits passing over any declaration in σ. The rule is written in a manner that introduces non-determinism, but in an implementation it should be complete to first check that the first declarations matching, and then, if not, fall through to this rule. (Conjecture: whether or not signature subsumption succeeds does not depend on which matching declaration you choose.)

$$\frac{\sigma \;\leq\; (\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L}\rhd\mathsf{r},\sigma \;\leq\; (\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}$$

References into Σ must agree:

$$\frac{\Sigma_\rho \vdash \mathsf{x}\equiv\mathsf{y}:\_ \quad \sigma \;\leq\; (\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L}\rhd\mathsf{y},\sigma \;\leq\; (\mathsf{D}\,;\,\mathsf{L}\rhd\mathsf{x},\delta)\,|\,\mathsf{D}';\mathsf{L}\rhd\mathsf{y},\sigma'}$$

$$\frac{\mathsf{block}\,\mathsf{x}=\mathsf{bs},\mathsf{block}\,\mathsf{y}=\mathsf{bs}'\text{ in }\Sigma_\rho \quad \Sigma_\rho\vdash\mathsf{bs}\equiv\mathsf{bs}'\,\mathsf{blockspec} \quad \sigma \;\leq\; (\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L}\rhd\mathsf{y},\sigma \;\leq\; (\mathsf{D}\,;\,\mathsf{L}\rhd\mathsf{x},\delta)\,|\,\mathsf{D}';\mathsf{L}\rhd\mathsf{y},\sigma'}$$

$$\frac{\mathsf{x}:\%\alpha,\mathsf{y}:\%\alpha'\text{ in }\Sigma_\rho,\Sigma_\omega \quad \Sigma_\rho\vdash\%\alpha\equiv\%\alpha'\,\mathsf{dir} \quad \sigma \;\leq\; (\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L}\rhd\mathsf{y},\sigma \;\leq\; (\mathsf{D}\,;\,\mathsf{L}\rhd\mathsf{x},\delta)\,|\,\mathsf{D}';\mathsf{L}\rhd\mathsf{y},\sigma'}$$

For references into D, the corresponding component of Σ must suffice, and the rules substitute its name in the inductive call to propagate the equality:

$$\frac{\sigma_1 \;\leq\; (\mathsf{D}\,;\,\delta_1)\,|\,\mathsf{D}_1;\sigma_1' \quad \sigma_2 \;\leq\; (\mathsf{D}_1\,;\,\delta_2)\,|\,\mathsf{D}_2;\sigma_2'}{\mathsf{l}\rhd\sigma_1,\sigma_2 \;\leq\; (\mathsf{D}\,;\,\mathsf{l}\rhd\delta_1,\delta_2)\,|\,\mathsf{D}_2;\mathsf{l}\rhd\sigma_1',\sigma_2'}$$

$$\frac{\mathsf{y}:\mathsf{e}'\text{ in }\Sigma_\rho \quad \Sigma_\rho\vdash\mathsf{e}'\equiv\mathsf{e}\,\mathsf{class} \quad \sigma \;\leq\; ([\mathsf{y}/\mathsf{x}]\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L}\rhd\mathsf{y},\sigma \;\leq\; (\mathsf{l}'\rhd\mathsf{x}:\mathsf{e}.\,\mathsf{D}\,;\,\mathsf{L}\rhd\mathsf{l}',\delta)\,|\,\mathsf{D}';\mathsf{L}\rhd\mathsf{y},\sigma'}$$

$$\frac{\mathsf{y}:\mathsf{e}'=_\mathsf{a}\mathsf{e}''\text{ in }\Sigma_\rho \quad \Sigma_\rho\vdash\mathsf{e}'\equiv\mathsf{e}\,\mathsf{class} \quad \sigma \;\leq\; ([\mathsf{y}/\mathsf{x}]\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L}\rhd\mathsf{y},\sigma \;\leq\; (\mathsf{l}'\rhd\mathsf{x}:\mathsf{e}.\,\mathsf{D}\,;\,\mathsf{L}\rhd\mathsf{l}',\delta)\,|\,\mathsf{D}';\mathsf{L}\rhd\mathsf{y},\sigma'}$$

$$\frac{\mathsf{y}:\mathsf{e}''=_{\mathsf{a}'}\mathsf{e}'''\text{ in }\Sigma_\rho \quad \Sigma_\rho\vdash\mathsf{e}\equiv\mathsf{e}''\,\mathsf{class} \quad \Sigma_\rho\vdash\mathsf{e}'\equiv\mathsf{e}''':\mathsf{e} \quad \sigma \;\leq\; ([\mathsf{y}/\mathsf{x}]\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L}\rhd\mathsf{y},\sigma \;\leq\; (\mathsf{l}'\rhd\mathsf{x}:\mathsf{e}=_\mathsf{a}\mathsf{e}'.\,\mathsf{D}\,;\,\mathsf{L}\rhd\mathsf{l}',\delta)\,|\,\mathsf{D}';\mathsf{L}\rhd\mathsf{y},\sigma'}$$

$$\frac{\mathsf{block}\,\mathsf{y} = \mathsf{bs}\ \mathsf{in}\ \Sigma_\rho \quad \sigma \ \leq\ ([\mathsf{y}/\mathsf{x}]\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L} \rhd \mathsf{y}, \sigma \ \leq\ (\mathsf{block}\,\mathsf{l}' \rhd \mathsf{x}.\ \mathsf{D}\,;\,\mathsf{L} \rhd \mathsf{l}', \delta)\,|\,\mathsf{D}';\mathsf{L} \rhd \mathsf{y}, \sigma'}$$

$$\frac{\mathsf{block}\,\mathsf{y} = \mathsf{bs}'\ \mathsf{in}\ \Sigma_\rho \quad \Sigma_\rho \vdash\ \mathsf{bs} \equiv \mathsf{bs}'\ \mathsf{blockspec} \quad \sigma \ \leq\ ([\mathsf{y}/\mathsf{x}]\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L} \rhd \mathsf{y}, \sigma \ \leq\ (\mathsf{block}\,\mathsf{l}' \rhd \mathsf{x} = \mathsf{bs}.\ \mathsf{D}\,;\,\mathsf{L} \rhd \mathsf{l}', \delta)\,|\,\mathsf{D}';\mathsf{L} \rhd \mathsf{y}, \sigma'}$$

$$\frac{\mathsf{y} : \%\alpha'\ \mathsf{in}\ \Sigma_\rho, \Sigma_\omega \quad \Sigma_\rho \vdash\ \%\alpha \equiv \%\alpha'\ \mathsf{dir} \quad \sigma \ \leq\ ([\mathsf{y}/\mathsf{x}]\mathsf{D}\,;\,\delta)\,|\,\mathsf{D}';\sigma'}{\mathsf{L} \rhd \mathsf{y}, \sigma \ \leq\ (\mathsf{l}' \rhd \%\alpha.\ \mathsf{D}\,;\,\mathsf{L} \rhd \mathsf{l}', \delta)\,|\,\mathsf{D}';\mathsf{L} \rhd \mathsf{y}, \sigma'}$$

$\boxed{\Sigma_\rho; \Sigma_\omega; \sigma \vdash \mathsf{N} : (\mathsf{D}, \delta) \Rightarrow (\mathsf{D}', \delta')}$ In the rules, we elide $\Sigma_\rho, \Sigma_\omega, \sigma$. Note that the rules use the signature append judgement in a non-deterministic fashion at an usual mode to extract a component of a signature.

$$\frac{\begin{array}{c}\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e} \\ \delta(\mathsf{p}) = \mathsf{l} \\ (\Sigma'_\rho\,;\,\Sigma'_\omega\ \sim\ \mathsf{D}_1)@(\mathsf{D}_2, \cdot) = (\mathsf{D}, \cdot) \\ \mathsf{D}_2 = \mathsf{l} \rhd \mathsf{x} : \mathsf{e}'.\ \mathsf{D}'_2 \\ \Sigma_\rho, \Sigma'_\rho \vdash\ \mathsf{e} : \mathsf{e}' \\ (\Sigma'_\rho\,;\,\Sigma'_\omega\ \sim\ \mathsf{D}_1)@(\mathsf{l} \rhd \mathsf{x} : \mathsf{e}' =_{\%\mathsf{abbrev}}\ \mathsf{e}.\ \mathsf{D}'_2, \cdot) = (\mathsf{D}', \cdot)\end{array}}{\mathsf{p} := \mathsf{e} : (\mathsf{D}, \delta) \Rightarrow (\mathsf{D}', \delta)}$$

$$\frac{\begin{array}{c}\sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e} \\ \delta(\mathsf{p}) = \mathsf{l} \\ (\Sigma'_\rho\,;\,\Sigma'_\omega\ \sim\ \mathsf{D}_1)@(\mathsf{D}_2, \cdot) = (\mathsf{D}, \cdot) \\ \mathsf{D}_2 = \mathsf{l} \rhd \mathsf{x} : \mathsf{e}'' =_{\mathsf{a}}\ \mathsf{e}'.\ \mathsf{D}'_2 \\ \Sigma_\rho, \Sigma'_\rho \vdash\ \mathsf{e} \equiv \mathsf{e}' : \mathsf{e}''\end{array}}{\mathsf{p} := \mathsf{e} : (\mathsf{D}, \delta) \Rightarrow (\mathsf{D}, \delta)}$$

$$\frac{\begin{array}{c}\sigma \vdash \mathsf{bs} \rightsquigarrow \mathsf{bs} \\ \delta(\mathsf{p}) = \mathsf{l} \\ (\Sigma'_\rho\,;\,\Sigma'_\omega\ \sim\ \mathsf{D}_1)@(\mathsf{D}_2, \cdot) = (\mathsf{D}, \cdot) \\ \mathsf{D}_2 = \mathsf{block}\,\mathsf{l} \rhd \mathsf{x}.\ \mathsf{D}'_2 \\ (\Sigma'_\rho\,;\,\Sigma'_\omega\ \sim\ \mathsf{D}_1)@(\mathsf{block}\,\mathsf{l} \rhd \mathsf{x} = \mathsf{bs}.\ \mathsf{D}'_2, \cdot) = (\mathsf{D}', \cdot)\end{array}}{\mathsf{p} := \mathsf{bs} : (\mathsf{D}, \delta) \Rightarrow (\mathsf{D}', \delta)}$$

$$\frac{\begin{array}{c}\sigma \vdash \mathsf{bs} \rightsquigarrow \mathsf{bs} \\ \delta(\mathsf{p}) = \mathsf{l} \\ (\Sigma'_\rho\,;\,\Sigma'_\omega \;\sim\; \mathsf{D}_1)@(\mathsf{D}_2, \cdot) = (\mathsf{D}, \cdot) \\ \mathsf{D}_2 = \mathsf{block}\,\mathsf{l} \rhd \mathsf{x} = \mathsf{bs}'.\ \mathsf{D}'_2 \\ \Sigma_\rho, \Sigma'_\rho \vdash \mathsf{bs} \equiv \mathsf{bs}'\ \mathsf{blockspec}\end{array}}{\mathsf{p} := \mathsf{bs} : (\mathsf{D}, \delta) \Rightarrow (\mathsf{D}, \delta)}$$

The following rule covers path instantiations for blocks and directives:

$$\frac{\begin{array}{c}\sigma(\mathsf{p}') = \mathsf{x}' \\ \delta(\mathsf{p}) = \mathsf{l} \\ \mathsf{D}|\mathsf{p} \rhd \mathsf{l} = \mathsf{D}'; \Sigma'_\rho; \Sigma'_\omega \\ \Sigma_\rho, \Sigma'_\rho; \Sigma_\omega, \Sigma'_\omega \vdash \mathsf{p}' \rhd \mathsf{x}' \;\leq\; (\mathsf{D}'\,;\,\mathsf{p} \rhd \mathsf{l})\,|\,\sigma' \\ (\mathsf{D}; \delta)[\mathsf{x}'/\mathsf{l}] = (\mathsf{D}''; \delta')\end{array}}{\mathsf{p} := \mathsf{p}' : (\mathsf{D}, \delta) \Rightarrow (\mathsf{D}'', \delta')}$$

$$\frac{\begin{array}{c}\sigma(\mathsf{p}') = \sigma' \\ \delta(\mathsf{p}) = \delta' \\ \mathsf{D}|\delta' = \mathsf{D}'; \Sigma'_\rho; \Sigma'_\omega \\ \Sigma_\rho, \Sigma'_\rho; \Sigma_\omega, \Sigma'_\omega \vdash \sigma' \;\leq\; (\mathsf{D}'\,;\,\delta')\,|\,\sigma'' \\ \sigma'' \,/\, \delta' = \beta' \\ (\mathsf{D}; \delta)[\beta'] = (\mathsf{D}''; \delta'')\end{array}}{\mathsf{p} := \mathsf{p}' : (\mathsf{D}, \delta) \Rightarrow (\mathsf{D}'', \delta'')}$$

$\boxed{\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi}$

$$\frac{}{\cdot \rightsquigarrow \cdot\,;\,\cdot\,;\,\cdot}$$

$$\frac{\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi \quad \Sigma\,;\cdot\,;\sigma \vdash_\psi \mathsf{S} \rightsquigarrow \mathsf{D}\,;\,\delta}{\Sigma, \mathsf{l} = \mathsf{S} \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi, \mathsf{l} = (\mathsf{D}, \delta)}$$

In the next two rules, the signature formation check handles freezing.

$$\frac{\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi \quad \sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e} \quad \Sigma \vdash \mathsf{x} : \mathsf{e}\,\mathsf{sig} \quad \mathsf{x}\,\mathsf{fresh}}{\Sigma, \mathsf{L} : \mathsf{e} \rightsquigarrow \Sigma, \mathsf{x} : \mathsf{e}\,;\,\sigma, \mathsf{L} \rhd \mathsf{x}\,;\,\psi}$$

$$\frac{\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi \quad \sigma \vdash \mathsf{e} \rightsquigarrow \mathsf{e} \quad \sigma \vdash \mathsf{e}' \rightsquigarrow \mathsf{e}' \quad \Sigma \vdash \mathsf{x} : \mathsf{e} =_{\mathsf{a}} \mathsf{e}'\ \mathsf{sig} \quad \mathsf{x}\ \mathsf{fresh}}{\Sigma, \mathsf{L} : \mathsf{e} =_{\mathsf{a}} \mathsf{e}' \rightsquigarrow \Sigma, \mathsf{x} : \mathsf{e} =_{\mathsf{a}} \mathsf{e}'\,;\,\sigma, \mathsf{L} \rhd \mathsf{x}\,;\,\psi}$$

$$\frac{\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi \quad \sigma \vdash \mathsf{bs} \rightsquigarrow \mathsf{bs} \quad \Sigma \vdash \mathsf{bs}\ \mathsf{blockspec} \quad \mathsf{x}\ \mathsf{fresh}}{\Sigma, \mathsf{block}\ \mathsf{l} = \mathsf{bs} \rightsquigarrow \Sigma, \mathsf{block}\ \mathsf{x} = \mathsf{bs}\,;\,\sigma, \mathsf{l} \rhd \mathsf{x}\,;\,\psi}$$

$$\frac{\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi \quad \sigma \vdash \%\alpha \rightsquigarrow \%\alpha \quad \Sigma \vdash \%\alpha\ \mathsf{succeeds} \quad \mathsf{x}\ \mathsf{fresh}}{\Sigma, \mathsf{L} : \%\alpha \rightsquigarrow \Sigma, \mathsf{x} : \%\alpha\,;\,\sigma, \mathsf{L} \rhd \mathsf{x}\,;\,\psi}$$

$$\frac{\begin{array}{c}\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi \\ \Sigma\,;\,\cdot\,;\,\sigma \vdash_{\psi} \mathsf{S} \rightsquigarrow \mathsf{D}\,;\,\delta \\ \mathsf{D};\delta \Rightarrow_t \Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma' \\ \mathsf{freezes}(\Sigma_\rho) = \Sigma_f \\ \Sigma \vdash \Sigma_\rho, \Sigma_\omega, \Sigma_f\ \mathsf{sig}\end{array}}{\Sigma, \mathsf{l} : \mathsf{S} \rightsquigarrow \Sigma, \Sigma_\rho, \Sigma_\omega, \Sigma_f\,;\,\sigma, \sigma'\,;\,\psi}$$

Σ_ρ must be checked here, rather than assumed to be well-formed, to catch freezing violations. In Section 5, we discuss methods of catching these violations sooner.

$$\frac{\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi \quad \Sigma\,;\,\cdot\,;\,\sigma \vdash_{\psi} \mathsf{S} \rightsquigarrow \mathsf{D}\,;\,\delta \quad \sigma(\mathsf{p}) = \sigma' \quad \Sigma;\cdot \vdash \sigma' \leq (\mathsf{D}\,;\,\delta)\,|\,\sigma''}{\Sigma, \mathsf{l} : \mathsf{S} = \mathsf{p} \rightsquigarrow \Sigma\,;\,\sigma, \mathsf{l} \rhd \sigma''\,;\,\psi}$$

$$\frac{\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi \quad \sigma(\mathsf{p}) = \sigma'}{\Sigma, \mathsf{open}\ \mathsf{p} \rightsquigarrow \Sigma\,;\,\sigma, \sigma'\,;\,\psi}$$

3.5 Definitions of Auxiliary Judgements

We now present the definitions of the auxiliary judgements. The reader may wish to recall the informal descriptions of these judgements in Section 3.3.

$\boxed{\mathsf{l} \# \delta}$ $\boxed{\delta \# \delta'}$

$$\frac{}{\_ \# \delta} \qquad \frac{}{\mathsf{l} \# \cdot} \qquad \frac{\mathsf{l} \neq \mathsf{L}}{\mathsf{l} \# \delta, \mathsf{L} \rhd \mathsf{r}}$$

We elide the definition of the second judgement.

$\boxed{\delta(\mathsf{l}) = \mathsf{r}}$

$$\frac{}{\delta, \mathsf{l} \rhd \mathsf{r}(\mathsf{l}) = \mathsf{r}} \qquad \frac{\delta(\mathsf{l}) = \mathsf{r} \quad \mathsf{L}' \neq \mathsf{l}}{\delta, \mathsf{L}' \rhd \mathsf{r}'(\mathsf{l}) = \mathsf{r}}$$

$\boxed{\delta(\mathsf{l}.\mathsf{p}) = \mathsf{r}}$

$$\frac{\delta(\mathsf{l}) = \mathsf{r}}{\delta(\mathsf{l}) = \mathsf{r}} \qquad \frac{\delta(\mathsf{l}) = \delta' \quad \delta'(\mathsf{p}) = \mathsf{r}}{\delta(\mathsf{l}.\mathsf{p}) = \mathsf{r}}$$

$\boxed{[\mathsf{x} \leftarrow \mathsf{l}]\delta = \delta'}$

$$\frac{[\mathsf{x} \leftarrow \mathsf{l}]\delta = \delta'}{[\mathsf{x} \leftarrow \mathsf{l}]\delta, \mathsf{L} \rhd \mathsf{x} = \delta', \mathsf{L} \rhd \mathsf{l}} \qquad \frac{\mathsf{x} \neq \mathsf{y} \quad [\mathsf{x} \leftarrow \mathsf{l}]\delta = \delta'}{[\mathsf{x} \leftarrow \mathsf{l}]\delta, \mathsf{L} \rhd \mathsf{y} = \delta', \mathsf{L} \rhd \mathsf{y}}$$

$$\frac{[\mathsf{x} \leftarrow \mathsf{l}]\delta = \delta'}{[\mathsf{x} \leftarrow \mathsf{l}]\delta, \mathsf{L} \rhd \mathsf{l}' = \delta', \mathsf{L} \rhd \mathsf{l}'} \qquad \frac{[\mathsf{x} \leftarrow \mathsf{l}]\delta_1 = \delta_1' \quad [\mathsf{x} \leftarrow \mathsf{l}]\delta_2 = \delta_2'}{[\mathsf{x} \leftarrow \mathsf{l}]\delta_1, \mathsf{L} \rhd \delta_2 = \delta_1', \mathsf{L} \rhd \delta_2'}$$

$\boxed{[\mathsf{l} \leftarrow \mathsf{x}]\delta = \delta'}$

$$\frac{[\mathsf{l} \leftarrow \mathsf{x}]\delta = \delta'}{[\mathsf{l} \leftarrow \mathsf{x}]\delta, \mathsf{L} \rhd \mathsf{l} = \delta', \mathsf{L} \rhd \mathsf{x}} \qquad \frac{\mathsf{l}' \neq \mathsf{l} \quad [\mathsf{l}' \leftarrow \mathsf{x}]\delta = \delta'}{[\mathsf{l} \leftarrow \mathsf{x}]\delta, \mathsf{L} \rhd \mathsf{l}' = \delta', \mathsf{L} \rhd \mathsf{l}'}$$

$$\frac{[\mathsf{l} \leftarrow \mathsf{x}]\delta = \delta'}{[\mathsf{l} \leftarrow \mathsf{x}]\delta, \mathsf{L} \rhd \mathsf{y} = \delta', \mathsf{L} \rhd \mathsf{y}} \qquad \frac{[\mathsf{l} \leftarrow \mathsf{x}]\delta_1 = \delta_1' \quad [\mathsf{l} \leftarrow \mathsf{x}]\delta_2 = \delta_2'}{[\mathsf{l} \leftarrow \mathsf{x}]\delta_1, \mathsf{L} \rhd \delta_2 = \delta_1', \mathsf{L} \rhd \delta_2'}$$

$\boxed{\mathsf{p} \rhd \mathsf{r} = \delta}$

$$\frac{}{\mathsf{l} \rhd \mathsf{r} = \mathsf{l} \rhd \mathsf{r}} \qquad \frac{\mathsf{p} \rhd \mathsf{r} = \delta}{\mathsf{l}.\mathsf{p} \rhd \mathsf{r} = \mathsf{l} \rhd \delta}$$

$\boxed{\mathsf{D}; \delta \rightrightarrows_s \Sigma_\rho ; \Sigma_\omega ; \sigma}$ where $s = m \mid t$ is a tag distinguishing being called from a module signature (m) from being called from a top-level signature (t).

$$\frac{}{\cdot; \sigma \rightrightarrows_s \cdot ; \cdot ; \sigma} \qquad \frac{\mathsf{x}\,\mathsf{fresh} \quad \mathsf{D}; [\mathsf{l} \leftarrow \mathsf{x}]\delta \rightrightarrows_s \Sigma_\rho ; \Sigma_\omega ; \sigma}{\mathsf{l} \rhd \mathsf{x} : \mathsf{e}.\ \mathsf{D}; \delta \rightrightarrows_s \mathsf{x} : \mathsf{e}, \Sigma_\rho ; \Sigma_\omega ; \sigma}$$

$$\frac{\mathsf{x}\,\mathsf{fresh} \quad \mathsf{D};[\mathsf{l}\leftarrow\mathsf{x}]\delta \rightrightarrows_s \Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma}{\mathsf{l}\triangleright\mathsf{x}:\mathsf{e}=_\mathsf{a}\mathsf{e}'.\,\mathsf{D};\delta \rightrightarrows_s \mathsf{x}:\mathsf{e}=_\mathsf{a}\mathsf{e}',\Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma}$$

The following rule only fires when called from a module signature; at the top level, we do not permit undefined blocks.

$$\frac{\mathsf{x}\,\mathsf{fresh} \quad \mathsf{D};[\mathsf{l}\leftarrow\mathsf{x}]\delta \rightrightarrows_m \Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma}{\mathsf{block}\,\mathsf{l}\triangleright\mathsf{x}.\,\mathsf{D};\delta \rightrightarrows_m \mathsf{block}\,\mathsf{x}=\mathsf{some}\,(\cdot)\,\mathsf{block}\,(\cdot),\Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma}$$

$$\frac{\mathsf{x}\,\mathsf{fresh} \quad \mathsf{D};[\mathsf{l}\leftarrow\mathsf{x}]\delta \rightrightarrows_s \Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma}{\mathsf{block}\,\mathsf{l}\triangleright\mathsf{x}=\mathsf{bs}.\,\mathsf{D};\delta \rightrightarrows_s \mathsf{block}\,\mathsf{x}=\mathsf{bs},\Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma}$$

$$\frac{\mathsf{x}\,\mathsf{fresh} \quad \%\alpha\,\mathsf{run} \quad \mathsf{D};[\mathsf{l}\leftarrow\mathsf{x}]\delta \rightrightarrows_s \Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma}{\mathsf{l}\triangleright\%\alpha.\,\mathsf{D};\delta \rightrightarrows_s \mathsf{x}:\%\alpha,\Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma}$$

$$\frac{\mathsf{x}\,\mathsf{fresh} \quad \%\alpha\,\mathsf{wait} \quad \mathsf{D};[\mathsf{l}\leftarrow\mathsf{x}]\delta \rightrightarrows_s \Sigma_\rho\,;\,\Sigma_\omega\,;\,\sigma}{\mathsf{l}\triangleright\%\alpha.\,\mathsf{D};\delta \rightrightarrows_s \Sigma_\rho\,;\,\mathsf{x}:\%\alpha,\Sigma_\omega\,;\,\sigma}$$

$$\boxed{(\Sigma_\rho\,;\,\Sigma_\omega \sim \mathsf{D}_1)@(\mathsf{D}_2,\delta_2)=(\mathsf{D}_{12},\delta_2')}$$

$$\frac{}{(\cdot\,;\,\cdot\sim\cdot)@(\mathsf{D}_2,\delta_2)=(\mathsf{D}_2,\delta_2)}$$

$$\frac{(\Sigma_\rho\,;\,\Sigma_\omega \sim \mathsf{D}_1)@(\mathsf{D}_2,\delta_2)=(\mathsf{D}_{12},\delta_2')}{(\mathsf{x}:\mathsf{e},\Sigma_\rho\,;\,\Sigma_\omega \sim \mathsf{l}\triangleright\mathsf{x}:\mathsf{e}.\,\mathsf{D}_1)@(\mathsf{D}_2,\delta_2)=(\mathsf{l}\triangleright\mathsf{x}:\mathsf{e}.\,\mathsf{D}_{12},[\mathsf{x}\leftarrow\mathsf{l}]\delta_2')}$$

$$\frac{(\Sigma_\rho\,;\,\Sigma_\omega \sim \mathsf{D}_1)@(\mathsf{D}_2,\delta_2)=(\mathsf{D}_{12},\delta_2')}{(\mathsf{x}:\mathsf{e}=_\mathsf{a}\mathsf{e}',\Sigma_\rho\,;\,\Sigma_\omega \sim \mathsf{l}\triangleright\mathsf{x}:\mathsf{e}=_\mathsf{a}\mathsf{e}'.\,\mathsf{D}_1)@(\mathsf{D}_2,\delta_2)=(\mathsf{l}\triangleright\mathsf{x}:\mathsf{e}=_\mathsf{a}\mathsf{e}'.\,\mathsf{D}_{12},[\mathsf{x}\leftarrow\mathsf{l}]\delta_2')}$$

$$\frac{(\Sigma_\rho\,;\,\Sigma_\omega \sim \mathsf{D}_1)@(\mathsf{D}_2,\delta_2)=(\mathsf{D}_{12},\delta_2')}{(\mathsf{block}\,\mathsf{x}=\mathsf{some}\,(\cdot)\,\mathsf{block}\,(\cdot),\Sigma_\rho\,;\,\Sigma_\omega \sim \mathsf{block}\,\mathsf{l}\triangleright\mathsf{x}.\,\mathsf{D}_1)@(\mathsf{D}_2,\delta_2)=(\mathsf{block}\,\mathsf{l}\triangleright\mathsf{x}.\,\mathsf{D}_{12},[\mathsf{x}\leftarrow\mathsf{l}]\delta_2')}$$

$$\frac{(\Sigma_\rho\,;\,\Sigma_\omega \;\sim\; \mathsf{D}_1)@(\mathsf{D}_2,\delta_2) = (\mathsf{D}_{12},\delta_2')}{(\mathsf{block}\,\mathsf{x}=\mathsf{bs},\Sigma_\rho\,;\,\Sigma_\omega \;\sim\; \mathsf{block}\,\mathsf{l}\rhd\mathsf{x}=\mathsf{bs}.\,\mathsf{D}_1)@(\mathsf{D}_2,\delta_2) = (\mathsf{block}\,\mathsf{l}\rhd\mathsf{x}=\mathsf{bs}.\,\mathsf{D}_{12},[\mathsf{x}\leftarrow\mathsf{l}]\delta_2')}$$

$$\frac{(\Sigma_\rho\,;\,\Sigma_\omega \;\sim\; \mathsf{D}_1)@(\mathsf{D}_2,\delta_2) = (\mathsf{D}_{12},\delta_2')}{(\mathsf{x}:\%\alpha,\Sigma_\rho\,;\,\Sigma_\omega \;\sim\; \mathsf{l}\rhd\%\alpha.\,\mathsf{D}_1)@(\mathsf{D}_2,\delta_2) = (\mathsf{l}\rhd\%\alpha.\,\mathsf{D}_{12},[\mathsf{x}\leftarrow\mathsf{l}]\delta_2')}$$

$$\frac{(\Sigma_\rho\,;\,\Sigma_\omega \;\sim\; \mathsf{D}_1)@(\mathsf{D}_2,\delta_2) = (\mathsf{D}_{12},\delta_2')}{(\Sigma_\rho\,;\,\mathsf{x}:\%\alpha,\Sigma_\omega \;\sim\; \mathsf{l}\rhd\%\alpha.\,\mathsf{D}_1)@(\mathsf{D}_2,\delta_2) = (\mathsf{l}\rhd\%\alpha.\,\mathsf{D}_{12},[\mathsf{x}\leftarrow\mathsf{l}]\delta_2')}$$

$\boxed{\mathsf{firstlab}(\delta)=\mathsf{l}}$

$$\frac{}{\mathsf{firstlab}(\mathsf{L}\rhd\mathsf{l},\delta)=\mathsf{l}} \qquad \frac{\mathsf{firstlab}(\delta)=\mathsf{l}}{\mathsf{firstlab}(\mathsf{L}\rhd\mathsf{x},\delta)=\mathsf{l}}$$

$$\frac{\mathsf{firstlab}(\delta)=\mathsf{l}}{\mathsf{firstlab}(\mathsf{L}\rhd\delta,\delta')=\mathsf{l}} \qquad \frac{\mathsf{firstlab}(\delta')=\mathsf{l}}{\mathsf{firstlab}(\mathsf{L}\rhd\sigma,\delta')=\mathsf{l}}$$

$\boxed{\mathsf{lastlab}(\delta)=\mathsf{l}}$

$$\frac{}{\mathsf{lastlab}(\delta,\mathsf{L}\rhd\mathsf{l})=\mathsf{l}} \qquad \frac{\mathsf{lastlab}(\delta)=\mathsf{l}}{\mathsf{lastlab}(\delta,\mathsf{L}\rhd\mathsf{x})=\mathsf{l}}$$

$$\frac{\mathsf{lastlab}(\delta)=\mathsf{l}}{\mathsf{lastlab}(\delta',\mathsf{L}\rhd\delta)=\mathsf{l}} \qquad \frac{\mathsf{lastlab}(\delta')=\mathsf{l}}{\mathsf{lastlab}(\delta',\mathsf{L}\rhd\sigma)=\mathsf{l}}$$

$\boxed{\mathsf{D}|\delta=\mathsf{D}';\Sigma_\rho;\Sigma_\omega}$

$$\frac{\begin{array}{c}\mathsf{firstlab}(\delta)=\mathsf{l}\quad\mathsf{lastlab}(\delta)=\mathsf{l}'\\(\Sigma_\rho\,;\,\Sigma_\omega \;\sim\; \mathsf{D}_1)@(\mathsf{D}_{23},\cdot)=(\mathsf{D},\cdot)\\\mathsf{D}_{23}=\mathsf{l}\rhd\mathsf{d}.\,\mathsf{D}'\\(\Sigma_\rho'\,;\,\Sigma_\omega' \;\sim\; \mathsf{D}_2)@(\mathsf{D}_3,\cdot)=(\mathsf{D}_{23},\cdot)\\\mathsf{D}_2=\ldots\mathsf{l}'\rhd\mathsf{d}'.\,\cdot\end{array}}{\mathsf{D}|\delta=\mathsf{D}_2;\Sigma_\rho;\Sigma_\omega}$$

$$\frac{}{\mathsf{D}|\sigma = \cdot;\cdot;\cdot}$$

$\boxed{\sigma \;/\; \delta = \beta}$ where $\beta \quad ::= \quad \cdot \mid \beta, \mathsf{x}/\mathsf{l}$

$$\frac{}{\cdot \;/\; \cdot = \cdot} \qquad \frac{\sigma \;/\; \delta = \beta}{\sigma \;/\; \mathsf{L} \rhd \mathsf{x}, \delta = \beta}$$

$$\frac{\sigma \;/\; \delta = \beta}{\mathsf{L} \rhd \mathsf{x}, \sigma \;/\; \mathsf{L} \rhd \mathsf{l}, \delta = \mathsf{x}/\mathsf{l}, \beta} \qquad \frac{\sigma \;/\; \delta = \beta \quad \sigma' \;/\; \delta' = \beta'}{\mathsf{l} \rhd \sigma, \sigma' \;/\; \mathsf{l} \rhd \delta, \delta' = \beta, \beta'}$$

$\boxed{(\mathsf{D}; \delta)[\beta] = (\mathsf{D}'; \delta')}$

In the rules, we abuse notation by abstracting the $\mathsf{l} \rhd \mathsf{x}$ part of the first component of a declaration D; for declarations that don't bind a variable (directives), assume a fresh x.

$$\frac{}{(\mathsf{D}; \delta)[\cdot] = (\mathsf{D}; \delta)} \qquad \frac{\mathsf{l} \neq \mathsf{l}' \quad (\mathsf{D}; \delta)[\mathsf{y}/\mathsf{l}', \beta] = (\mathsf{D}'; \delta')}{(\mathsf{l} \rhd \mathsf{x}.\ \mathsf{D}; \delta)[\mathsf{y}/\mathsf{l}', \beta] = (\mathsf{l} \rhd \mathsf{x}.\ \mathsf{D}'; \delta')}$$

$$\frac{([\mathsf{y}/\mathsf{x}]\mathsf{D}; [\mathsf{l} \leftarrow \mathsf{y}]\delta)[\beta] = (\mathsf{D}'; \delta')}{(\mathsf{l} \rhd \mathsf{x}.\ \mathsf{D}; \delta)[\mathsf{y}/\mathsf{l}, \beta] = (\mathsf{D}'; \delta')}$$

$\boxed{\mathsf{freezes}(\Sigma) = \Sigma'}$

$$\frac{}{\mathsf{freezes}(\cdot) = \cdot}$$

$$\frac{\Sigma, \mathsf{x} : \mathsf{e} \vdash \mathsf{x}\,\mathsf{famconst} \quad \mathsf{freezes}(\Sigma) = \Sigma' \quad \mathsf{y}\,\mathsf{fresh}}{\mathsf{freezes}(\Sigma, \mathsf{x} : \mathsf{e}) = \Sigma', \mathsf{y} : \mathsf{freeze}\,\mathsf{x}}$$

$$\frac{\Sigma, \mathsf{x} : \mathsf{e} \vdash \mathsf{x}\,\mathsf{famconst} \quad \mathsf{freezes}(\Sigma) = \Sigma' \quad \mathsf{y}\,\mathsf{fresh}}{\mathsf{freezes}(\Sigma, \mathsf{x} : \mathsf{e} =_{\mathsf{a}} \mathsf{e}') = \Sigma', \mathsf{y} : \mathsf{freeze}\,\mathsf{x}}$$

$$\frac{\text{not } (\Sigma, \mathsf{x} : \mathsf{e} \vdash \mathsf{x}\,\mathsf{famconst}) \quad \mathsf{freezes}(\Sigma) = \Sigma'}{\mathsf{freezes}(\Sigma, \mathsf{x} : \mathsf{e}) = \Sigma'}$$

$$\frac{\text{not } (\Sigma, \mathsf{x} : \mathsf{e} \vdash \mathsf{x}\,\mathsf{famconst}) \quad \mathsf{freezes}(\Sigma) = \Sigma'}{\mathsf{freezes}(\Sigma, \mathsf{x} : \mathsf{e} =_{\mathsf{a}} \mathsf{e}') = \Sigma'}$$

$$\frac{\mathsf{freezes}(\Sigma) = \Sigma'}{\mathsf{freezes}(\Sigma, \mathsf{block}\,\mathsf{x} = \mathsf{bs}) = \Sigma'}$$

$$\frac{\mathsf{freezes}(\Sigma) = \Sigma'}{\mathsf{freezes}(\Sigma, \mathsf{x} : \%\alpha) = \Sigma'}$$

4 Implementation

We have a partial implementation of the module system presented in the previous section. Although time limitations, technical issues interfacing with Twelf, and the demands of programming a moving target prevented us from completing the application, our experiences so far lead us to believe our formalism lends itself to a clean and understandable implementation. Our implementation interprets the judgments used in the elaboration as appropriately moded algorithms. Because the elaborative semantics have been rigorously defined in terms of inference rules, much of the code base is a straightforward transcription of the paper semantics.

The technical challenge that arose during our implementation stems from the fact that the definition of elaboration assumes that Twelf provides certain judgments about Twelf terms and directives. In our original proposal, our naive plan was to witness such properties by generating equivalent Twelf programs and running them through Twelf. However, there are some judgments such as $\Sigma \vdash \mathsf{bs} \equiv \mathsf{bs}'\ \mathsf{blockspec}$ for which it is not clear how to generate a witnessing Twelf program. The correct way to proceed is to interface with the Twelf implementation in order to write functions that witness the desired judgments. In order to do so, we must first answer unresolved technical questions as to how we would reconcile the functional view of Twelf implied by our semantics with the treatment of state in the actual Twelf implementation. Specifically, the order in which we check judgments under particular signatures does not necessarily align with how Twelf's state would evolve as we make our checks. To ensure correct behavior, we would need some mechanism for rolling back or swapping the execution state of the Twelf implementation.

```
s : type.

MOD = %sig a : s. %end.

Mod1 : MOD.    % OK
%freeze s.
Mod4 : MOD.    % NOT ALLOWED
```

Figure 7: Freezing and top-level families

5 Future Work and Conclusion

Though we believe that we have arrived at a reasonable checkpoint in the design of the module system, there are several extensions necessary to complete the project.

First, we have not yet rigorously proved any properties of the elaboration judgements. One desirable property is that the top-level judgement $\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi$ produces a valid Twelf signature Σ. We intend to define the judgement $\Sigma \vdash \Sigma'\,\mathsf{sig}$ in terms of the other Twelf judgements and verify that this invariant holds. Additionally, there are several internal invariants of the elaboration rules that could be verified. For example, the rules often consider a pair (D, δ) such that the labels in the range of δ are exactly those of the components of D. Though we used such invariants to guide our design, we have not yet formally stated and proved them.

Second, our elaboration judgements catch freezing violations relatively late. Specifically, as we noted in the definition of the judgement $\Sigma \rightsquigarrow \Sigma\,;\,\sigma\,;\,\psi$, the rule for top-level module declarations must check the signature Σ_ρ to ensure that it is well-formed. It would be preferable if this signature, which consists only of LF declarations and Twelf directives that are run while processing module signatures, were guaranteed to be well-formed by the elaboration. Indeed, we conjecture that the rules maintain this invariant for well-formedness in the LF sense (classifiers are well-formed; definands are of the appropriate type or kind). However, these signatures may trigger a freezing violation. We intend to revise the elaboration rules so that freezing violations are caught earlier. This extension will make it easier to state a precise invariant on the signatures Σ_ρ passed to the Twelf judgements during elaboration of module signatures.

A related issue concerns top-level family definitions: should it be permissible for constants in a module to extend top-level families? We conjecture that it should not, because allowing such extensions would interact

badly with catching freezing violations early. Specifically, it would be possible to define a top-level family s, then name a signature extending s, and then freeze s, which would invalidate the well-formedness of the named signature—later instantiations of that signature would cause a freezing violation. Figure 7 contains an example of such a scenario.

Fourth, in future work, it would be possible to extend the notion of signature matching presented here. We believe that the judgement $\Sigma_\rho; \Sigma_\omega \vdash \sigma \leq (\mathsf{D}; \delta) \mid \sigma'$ defines a useful notion of signature subsumption, where subsignatures may refine declarations with definitions and add additional fields. However, richer notions are possible: for example, we may consider allowing permuted fields (subset rather than subsequence) and implicit instantiation of Twelf's implicit Π-type and -kind parameters. As long as the supersignature δ does not contain any underscore fields, coercions witnessing these additional subsignature relationships can be written in the language by writing a module realizing δ in terms of σ using definitions. Underscore components are problematic because the programmer cannot refer to them in definitions. We do not yet know whether these limitations are problematic in practice.

Finally, once we have a working prototype, we will be able to write more involved examples and assess the design of the module language empirically.

In this report, we have presented the primary deliverable of our project proposal, the formal elaborative semantics of a module system for Twelf. Additionally, we have made significant progress toward a prototype implementation. In future work, we intend to pursue our other proposed goals, a complete prototype, a library of examples for demonstrating and evaluating the module system, and a real implementation in Twelf.

References

Andrew W. Appel. Foundational proof-carrying code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 247, Washington, DC, 2001. IEEE Computer Society.

Coq Development Team. The Coq proof assistant reference manual, 2006.

Karl Crary. Toward a foundational typed assembly language. In *Thirtieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, Louisiana, January 2003.

Rémy Haemmerlé and François Fages. Modules for prolog revisited. In *International Conference on Logic Programming*, pages 41–55, Seattle, WA, 2006.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40 (1), 1993.

Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998.

Florian Kammüller. Module reasoning in Isabelle. In David A. McAllester, editor, *International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 99–114, Pittsburgh, PA, 2000. Springer-Verlag.

Florian Kammüller, Markus Wenzel, and Lawrence Paulson. Locales: A sectioning concept for Isabelle. In *International Conference on Theorem Proving in Higher-Order Logics*, pages 149 – 166, Nice, France, 1999. Springer-Verlag.

Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Nice, France, 2007.

Dale Miller. A proposal for modules in Lambda-Prolog. In *ELP '93: Proceedings of the 4th International Workshop on Extensions of Logic Programming*, pages 206–221, St. Andrews, UK, 1994. Springer-Verlag. ISBN 3-540-58025-5.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *IEEE Symposium on Logic in Computer Science*, pages 312–322, Pacific Grove, CA, 1989.

Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *International Conference on Automated Deduction*, pages 202–206, 1999.