Principal Type Schemes for Modular Programs

Derek Dreyer and Matthias Blume

Toyota Technological Institute at Chicago
{dreyer,blume}@tti-c.org

Abstract. Two of the most prominent features of ML are its expressive module system and its support for Damas-Milner type inference. However, while the foundations of both these features have been studied extensively, their interaction has never received a proper type-theoretic treatment. One consequence is that both the official Definition and the alternative Harper-Stone semantics of Standard ML are difficult to implement correctly. To bolster this claim, we offer a series of short example programs on which no existing SML typechecker follows the behavior prescribed by either formal definition. It is unclear how to amend the implementations to match the definitions or vice versa. Instead, we propose a way of defining how type inference interacts with modules that is more liberal than any existing definition or implementation of SML and, moreover, admits a provably sound and complete typechecking algorithm via a straightforward generalization of Algorithm $\mathcal{W}$. In addition to being conceptually simple, our solution exhibits a novel hybrid of the Definition and Harper-Stone semantics of SML, and demonstrates the broader relevance of some type-theoretic techniques developed recently in the study of recursive modules.

1 Introduction

The standard way of defining the static semantics of languages with type inference is to give a set of *declarative* Curry-style typing rules that assign types to untyped terms. Since these rules are typically written in a nondeterministic fashion, it is not *prima facie* decidable whether a term can be assigned a type. The most canonical example of such a type system is that of Hindley and Milner (HM) [15], which is at the core of higher-order, typed languages like ML and Haskell. For HM, there exists a well-known type inference algorithm, Damas and Milner's Algorithm $\mathcal{W}$ [1], that is sound and complete with respect to it. Soundness means that if type inference succeeds, it returns a valid typing. Completeness means that if a valid typing exists, then type inference will succeed. To achieve completeness, Algorithm $\mathcal{W}$ relies on the fact that, in HM, all well-typed terms e have a *principal type scheme*, *i.e.,* a type τ that is in some sense more general than any other type that e could be assigned.

The Definition of Standard ML [16] (SML for short) provides a declarative semantics of the language, including both the core language and the module system. It is often assumed that this combination can be typechecked effectively via a straightforward extension of Algorithm $\mathcal{W}$. In this paper we observe that

```
functor F (X: sig type t end) = struct
    val f = id id
end
structure A = F (struct type t = int end)
structure B = F (struct type t = bool end)
```

Fig. 1. Common preamble for examples (a) and (b).

```
val _ = A.f 10
val _ = B.f false
```

Fig. 2. Example (a).

```
val _ = A.f 10
val _ = B.f "dude"
```

Fig. 3. Example (b).

this is not so. As we will explain, all existing implementations of SML differ from the Definition—and from one another—in rather subtle ways when it comes to the interaction of type inference and modules. In particular, this occurs when the *value restriction* forces certain module-level **val**-bindings to be monomorphic.

The value restriction [11, 22], which is important for ensuring type safety, limits the set of **val**-bindings that may be assigned polymorphic types to those whose right-hand sides are *syntactic values*, i.e., terms that are syntactically known to be free of effects. For example, consider

```
val f = id id
```

where `id` is the identity function **fn** `x => x`. According to the declarative semantics of SML, `f` can be assigned any (monomorphic) type $\tau \to \tau$ where τ is a type that is well-formed at the point where `f` is defined. However, it *cannot* be given the polymorphic type $\forall\alpha.\alpha \to \alpha$ because `id id` is not a syntactic value.

Example (a). The situation changes if `f` is defined in the body of a functor. Consider the code in Figures 1 and 2. The value restriction forces the type of `f` in functor `F` to be monomorphic. Since `A` is an instantiation of `F`, and since `A.f` is applied to an integer, one might expect that the type of `f` must be `int` $\to$ `int`, in which case the application of `B.f` to `false` would be ill-typed. In fact, though, there exists another solution: `f` can also be given the type `X.t` $\to$ `X.t`, in which case the type of `A.f` is `int` $\to$ `int`, the type of `B.f` is `bool` $\to$ `bool`, and the program typechecks. In essence, by appearing in the body of a functor with an abstract type argument, `f` can turn into an (explicit) polymorphic function.

This example has profound implications for the use of Damas-Milner type inference in typechecking modular programs. When `f` is typechecked by Algorithm $\mathcal{W}$, the algorithm returns a principal type scheme $\boldsymbol{\alpha} \to \boldsymbol{\alpha}$, where $\boldsymbol{\alpha}$ is a *unification variable* (u-var) that may be instantiated to a particular type later on. However, when the typechecker leaves the body of functor `F`, the u-var $\boldsymbol{\alpha}$ must become a Skolem function parameterized over the abstract type components of `F`'s argument, in this case the type `t`. When `A.f` is applied to `int`, we learn *not* that $\boldsymbol{\alpha}$ = `int`, but rather that $\boldsymbol{\alpha}$(`int`) = `int`. Similarly, when `B.f` is applied to `false`, we learn that $\boldsymbol{\alpha}$(`bool`) = `bool`. Together, these two equations determine that the only possible instantiation of $\boldsymbol{\alpha}$ is the identity function λ`t.t`.

In general, finding solutions to these kinds of constraints requires a form of higher-order unification. While it is possible that the fragment of higher-order unification required is decidable, no SML typechecker implements it. As a result, no existing SML implementation accepts Example (a)... that is, except MLton.

1.1 Generalized Functor Signatures

The MLton compiler for SML takes an unusual approach to the typechecking of programs with functors. Although it is well-known that MLton is a whole-program compiler that achieves great performance gains through defunctorization, it is perhaps less well-known that MLton performs defunctorization *during typechecking*. That is, after typechecking a functor such as `F` in Example (a), MLton inlines the definition of `F` at every point where `F` is applied before proceeding to typecheck the rest of the program. This has the effect that the definition of `F` is re-typechecked at every application, and each copy of `F` may be assigned a different signature. In the case of Example (a), this means that in the first copy of `F`, its binding for `f` may be assigned type `int` $\rightarrow$ `int`, and in the second copy of `F`, its binding for `f` may be assigned type `bool` $\rightarrow$ `bool`.

Example (b). While MLton's approach to typechecking functors results in the acceptance of Example (a), it also results in the acceptance of similar programs that are *not* well-typed according to the Definition. Consider Example (b) in Figures 1 and 3. Since `B.f` is now applied to a string instead of a boolean, there is no single type for `f` that would make the program typecheck. Put another way, there is no solution to the unification problem $\boldsymbol{\alpha}$(`int`) = `int` $\wedge$ $\boldsymbol{\alpha}$(`bool`) = `string`. However, since MLton inlines `F` prior to typechecking the definition of `B`, it is happy to assign `f` the type `string` $\rightarrow$ `string` in the second copy of `F`.

We prefer MLton's behavior to that prescribed by the Definition for several reasons. First, it does not require any higher-order unification and is therefore simpler and easier to implement. Second, it is more liberal than the Definition in a way that is perfectly type-safe. Third, it is arguably more intuitive. Given that the type `X.t` and the definition of `f` are completely unrelated, we feel it is very odd that Example (a) type-checks under the Definition but Example (b) does not. That said, the MLton approach has the serious drawback that it needs to know all uses of `F`, *i.e.,* it needs access to the whole program.

To overcome this limitation of MLton's approach, we observe that MLton's inlining of functors is analogous to the well-known explanation of **let**-polymorphism (*e.g.,* see Pierce's textbook [19]), in which **let** $x = e_1$ **in** e_2 is well-typed if and only if e_1 and $e_2[e_1/x]$ are. Inlining the definition of x has the same effect as binding x in the context with a generalized polymorphic type for e_1. Similarly, to mimic the inlining of a functor `F`, we need to bind `F` in the context with a *generalized functor signature*, *i.e.,* a signature that takes *implicit* type arguments in addition to the usual explicit module arguments. In the case of Examples (a) and (b), we would like to assign `F` the signature

`(X: sig type t end)` $\rightarrow$ $\underline{\forall\alpha.}$ **sig val `f`** : $\alpha \rightarrow \alpha$ **end**

At each application of `F`, the type argument α could then be implicitly instantiated with a new type τ, thus enabling both Examples (a) and (b) to typecheck.

```
functor G () = struct
    datatype t = V
    val f = id id
end
structure C = G()
val _ = C.f C.V
```

Fig. 4. Example (c).

```
functor G () = struct
    val f = id id
    datatype t = V
end
structure C = G()
val _ = C.f C.V
```

Fig. 5. Example (d).

The reason that it is sound to permit this kind of implicit polymorphic generalization at the definition of `F`—*i.e.,* the reason it does not violate the value restriction—is that functor bindings *are* bindings of syntactic values.

1.2 Abstract Data Types and Dependencies

The idea of generalized functor signatures sketched above is at the heart of the type system we present in Sections 2 and 3. However, functors are not the only complication that modular ML programs introduce into the HM type system. Another such complication is ML's facility for defining abstract data types.

Example (c). Consider the code in Figure 4. In this case, the body of functor `G` defines an abstract data type `t`, as well as a value `V` of type `t`, prior to its binding for `f`. Consequently, the application of `C.f` to `C.V` is well-typed according to the Definition, since `f` could have been assigned the type `t` $\rightarrow$ `t`.

We run into an interesting problem, though, if we attempt to typecheck this example using a generalized functor signature for `G`. In particular, the obvious generalized signature that one would expect to assign to `G` is the following:

$$\texttt{()} \rightarrow \underline{\forall\alpha.}\ \textbf{sig datatype t = V val f : } \alpha \rightarrow \alpha\ \textbf{end}$$

In order to typecheck the definition of `C` in such a way that the subsequent application (`C.f C.V`) will be well-typed, the application of `G` must instantiate `G`'s implicit parameter α with the type `C.t`. But `C.t` is not in scope until after `G` has been applied, so how can `C.t` be passed as an argument to `G`?!

One way to view this problem is as a variation on the original problem that we observed with typechecking Example (a). In parameterizing `G`'s signature over α, we failed to account for the possibility that α might refer to `t`. In other words, one might argue, the parameter α should really be a Skolem function, and `f`'s type should be $\alpha(\texttt{t}) \rightarrow \alpha(\texttt{t})$. This observation is not very encouraging, though, since it only seems to lead us back to the need for higher-order unification. Fortunately, as we will explain shortly in Section 2, we have an alternative solution to this dilemma that does not require higher-order unification.

Example (d). Lastly, let us consider the code in Figure 5, which is the same as that in Figure 4 save that the order of the bindings of `t` and `f` has been switched. Because of this switch, Example (d) is not legal SML, for `t` is not in scope at the point where `f` is defined.

The fact that the Definition treats Examples (c) and (d) differently means that a faithful implementation of SML must track the potential dependencies

Example	Definition	Reject All	No-HOU/No-track	MLton	Our Approach
(a)	✓	×	×	✓	✓
(b)	×	×	×	✓	✓
(c)	✓	×	✓	✓	✓
(d)	×	×	✓	×	✓

Fig. 6. Comparison of behaviors of different semantics and implementations on Examples (a)-(d). *Definition:* This reflects the behavior of both the Definition and the Harper-Stone alternative semantics of SML [7]. *Reject All:* SML/NJ, the ML Kit, TILT, SML.NET, and Hamlet reject all examples. *No-HOU/No-track*: Poly/ML, Alice, and Moscow ML fail to accept Example (a) and fail to reject Example (d). (Actually, Moscow ML rejects (d) in batch mode but accepts it in interactive mode.) *MLton* fails to reject Example (b). *Our Approach* is more liberal than any of the existing definitions or implementations, and it is easy to implement correctly.

between unification variables (u-vars) and the abstract types that are in scope when the u-vars are introduced. This tracking adds a layer of complexity to the type inference algorithm that, when combined with the problems we have observed concerning type inference and functors, seems to be tricky to get right. As evidence of this, we note that, with the exception of MLton (and Moscow ML when run in batch mode), no implementation of SML correctly handles both Examples (c) and (d) according to the Definition (see Figure 6). Furthermore, as the design we propose below will demonstrate, having a declarative semantics that permits Example (d) to typecheck is not only perfectly type-safe—it makes the typechecking algorithm much easier to specify.

1.3 The SML/NJ Approach

As we have seen, it is difficult to implement type inference for SML correctly. A number of existing implementations reject all four examples presented above—even though Examples (a) and (c) are legal SML—and for at least one compiler (namely, SML/NJ), the rejection of these examples is the result of a deliberate implementation decision [13]. Specifically, SML/NJ's policy is that no unification variables created during type inference are permitted to escape to the module level, even if subsequent code determines how they must be instantiated. This policy has the benefit of being consistent and predictable, and since programmers are not exactly clamoring for the rather contrived Examples (a)-(d) to be accepted anyway, one may wonder why it has not been adopted more widely.

In fact, there are several reasons why we find SML/NJ's *reject-all* approach unsatisfactory. First and foremost, a consequence of this policy that is well-known to be irritating to many programmers is that one cannot write a side-effecting module binding **val** `x = ref nil` and have the SML/NJ typechecker infer the specific type of `x` from later use within the module—one is forced to write a type annotation on `x`.

Second, SML/NJ's semantics is based on the *algorithmic* notion of unification variables escaping to the module level. In order to give a declarative account of SML/NJ's semantics, the typing rule for a module-level **val**-binding would have

to demand that the expression being bound have a *unique* type in the case that it is not a syntactic value. Since uniqueness is a higher-order statement about the set of all possible typing derivations for a given term, formulating such a rule requires care in order to ensure that the typing judgment is well-founded.

We believe it is straightforward to show that such a higher-order rule makes sense, *provided* that the static semantics of the core language does not depend on that of the module language (*i.e.,* that module definitions cannot appear within core terms). This assumption is valid for Standard ML. It is also true for Extended ML [8], in whose formalization Kahrs *et al.* employ similar higher-order rules with different motivation. However, from a language design standpoint, this condition is unnecessarily limiting. Several implementations of Standard ML (*e.g.,* Moscow ML and Alice ML) extend the language with features such as first-class modules [21] and the ability to write module bindings within **let**-expressions, which introduce interdependencies between the core and module languages. It is unclear whether the natural declarative account of SML/NJ's semantics is well-founded in the presence of such extensions.

2 Our Approach

Instead of attempting to prohibit any interaction between core type inference and module type checking, we propose a way of understanding and defining the semantics of type inference in the presence of modules that is *more* liberal than any existing definition or implementation of SML. In fact, our formalization of type inference (Section 3) embraces the interaction with modules in the sense that polymorphic generalization and instantiation (typically viewed as strictly core-language notions) are treated as coercions between the core and module languages. Moreover, our approach admits a provably sound and complete type-checking algorithm via a straightforward generalization of Algorithm $\mathcal{W}$. This is the first (positive) result that we are aware of concerning the interaction of ML-style modules and Damas-Milner type inference.

One key element of our approach is the idea of classifying functors using *generalized functor signatures* (GFS's), which we sketched at the end of Section 1. We will characterize their semantics precisely in Section 3.

The second key element of our approach concerns the treatment of abstract data types. As explained in Section 1, the problem with Example (c) is that we need access to the abstract type `C.t`, generated by the functor application `G()`, *ahead of time* so that we can use it to instantiate `G`'s implicit type argument. To make this possible, we use ideas and formal techniques from a type system developed recently by Dreyer [2] (in the study of recursive modules) that provides precisely the feature we are looking for: *forward references* to abstract types.

Traditionally, abstract data types are modeled by values of existential type $(\exists\alpha.\tau)$, which must be "unpacked" in order to obtain a fresh abstract type α and a value x of type τ representing the (operations on) values of type α [17]. In Dreyer's system, the type name α may be created ahead of time, before the package defining α and the (operations on) values of type α is even available. This is motivated by the goal of modeling recursive module programming, in which the abstract type components of a module may be "forward-declared".

To see how Dreyer's approach is useful in typechecking Example (c), let us first consider the declarative module typing judgment that we will formalize in Section 3. This judgment has the form $\Delta\,;\Gamma \vdash \mathit{mod} : \Sigma$ with $\overline{\alpha}\downarrow$, and can be read: "In type context Δ and term/module context Γ, module *mod* can be assigned signature Σ and, when evaluated, will define the abstract types $\overline{\alpha}$."[1] (*Note:* We use $\overline{\alpha}$ as a semantic representation of the abstract types defined by *mod*—the $\overline{\alpha}$ are not permitted to appear syntactically in *mod* itself.) An invariant of this judgment is that the variables $\overline{\alpha}$ must be bound in the type context Δ—*i.e.,* they must already have been created prior to the evaluation of *mod*. In order to ensure that abstract types are defined *exactly* once, we follow Dreyer in employing techniques reminiscent of a linear type system. In particular, we have several different binding forms for type variables that indicate whether or not they have been defined. In the typing judgment given above, $\overline{\alpha}$ are assumed to be bound in Δ as *undefined* (written $\overline{\alpha\uparrow \mathrm{K}}$), and the evaluation of *mod* has the effect of changing the bindings of $\overline{\alpha}$ to *defined* (written $\overline{\alpha\downarrow \mathrm{K}}$).

As for Example (c): Under the approach to declarative module typing we have sketched above, since the binding **structure** `C = G()` results in the definition of an abstract type `C.t`, the typing of this binding must occur in a context where `C.t`, represented semantically by some α, is already bound (as undefined). Since α must appear in the context of the binding, it is no problem to instantiate `G`'s implicit argument using α, and thus Example (c) will be deemed well-typed.

Concerning Example (d): As we argued in Section 1, we believe it is perfectly legitimate for Example (d) to be accepted, and our declarative module typing judgment affirms this stance. Specifically, since the body of functor `G`—let's call it *mod*—defines an abstract type `t`, it must be that the semantic type variable α representing `t` is bound as undefined in the initial context under which *mod* is typechecked. As a result, α is in scope throughout all of *mod*, including the binding for `f`, regardless of the order in which `t` and `f` are bound. (Admittedly, this has the somewhat odd effect that `f` is assigned a type with which the programmer could not have annotated `f` explicitly. However, due to the so-called *avoidance problem* [12], this situation already arises in SML in other contexts.)

In summary, the main benefit of using this style of declarative typing judgment is that we are freed from worrying about the relative order in which unification variables and ADT's are introduced into scope during type inference.

Finally, since our approach to module typing is based on a *type system*, which has been proven type-safe by Dreyer using standard syntactic methods, it is quite easy to show that our approach is type-safe. Following Harper and Stone [7], we do not give a dynamic semantics directly for our ML-style module language, but rather by *elaboration* (aka *evidence translation*) into an internal language type system (IL). In the case of the Harper-Stone alternative formalization of SML, that IL is a variant of Harper-Lillibridge/Leroy's module type system [6, 10]. The IL we employ here is a variant of Dreyer's type system for recursive modules (minus the recursion) [3]. The details of this translation are given in a companion technical report [4]. We omit further discussion due to space limitations.

[1] We adopt the notation $\overline{E}$ to mean a (possibly empty) ordered list $E_1, \ldots, E_n$.

Label Seq's	ℓs	$::= \epsilon \mid \ell.\ell s$
Paths	P	$::= \mathrm{X}.\ell s$
Kinds	K, L	$::= \mathbf{T} \mid \mathbf{T}^n \to \mathbf{T}$
Type Con's	*con*, *typ*	$::= \mathrm{P} \mid \alpha \mid typ \to typ \mid \lambda(\overline{\alpha}).typ \mid con(\overline{typ})$
Terms	*exp*	$::= \mathrm{P} \mid x \mid \lambda x.exp \mid exp_1(exp_2) \mid exp : typ \mid \mathtt{let}\ \mathrm{X} = mod\ \mathtt{in}\ exp$
Values	*val*	$::= \mathrm{P} \mid x \mid \lambda x.exp \mid val : typ$
Signatures	*sig*	$::= [\![\mathrm{K}]\!] \mid [\![= con : \mathrm{K}]\!] \mid [\![\forall(\overline{\alpha}).typ]\!] \mid [\![\overline{\ell \triangleright \mathrm{X} : sig}]\!] \mid (\mathrm{X} : sig_1) \to sig_2$
Modules	*mod*	$::= \mathrm{P} \mid [con] \mid [exp] \mid [\overline{\ell \triangleright \mathrm{X} = mod}] \mid \lambda(\mathrm{X} : sig).mod \mid \mathrm{P}_1(\mathrm{P}_2) \mid$ $\mathtt{let}\ \mathrm{X} = mod_1\ \mathtt{in}\ mod_2 \mid mod\ \mathtt{:>}\ sig \mid mod : sig$

Fig. 7. External language syntax

Our ability to prove type safety in a straightforward manner is a clear advantage of our approach over the *ad hoc* formal approach adopted by the Definition [16], as well as improvements to the Definition style such as Russo's [20]. As we will see in the next section, however, there are several aspects of our declarative typing judgment that are more reminiscent of the Definition than they are of the Harper-Stone semantics. Our design thus exhibits a viable hybrid of two approaches to defining SML that are often viewed as incompatible.

3 Declarative Semantics

Figure 7 presents the syntax of our *external* (*i.e.,* source-level) SML-like language. In order to provide a clean formal account of the essential issues, our formalism pares away some of the syntactic complexities of real SML programs.

First, we model type and value bindings in modules as a special case of module bindings, in which the module being bound is an *atomic* type or term module, *i.e.,* a module containing a single type or term component. The signature $[\![\mathrm{K}]\!]$ models an *opaque specification* of an atomic type module $[con]$ whose type (constructor) component *con* has kind K. The signature $[\![= con : \mathrm{K}]\!]$ models a *transparent specification* of an atomic type module whose type component is manifestly equal to *con* of kind K. The signature $[\![\forall(\overline{\alpha}).typ]\!]$ models a *value specification*, classifying atomic term modules $[exp]$ whose term component *exp* has type $\forall(\overline{\alpha}).typ$. (Note that if $\overline{\alpha} = \emptyset$, this degenerates from a polymorphic type to a monomorphic type.)

Second, we follow Harper and Lillibridge [6] and model structures as records $[\overline{\ell \triangleright \mathrm{X} = mod}]$, each of whose components has both a distinct external name (a label ℓ) and a distinct internal name (a variable X). The label is used to refer to the component from the outside of the module, whereas the variable is used to refer to the component in subsequent bindings within the structure. Thus, each X is bound in the context of the subsequent bindings and may be alpha-varied. For simplicity, we adopt the ML convention that projections are not permitted from arbitrary modules. The only projection form is the path P, which consists of zero or more projections from a module variable X. (Note: We will usually drop the trailing "$.\epsilon$" from a path, *e.g.,* writing X instead of $\mathrm{X}.\epsilon$.)

Third, we model the classic distinction between "polytypes" and "monotypes" as a special case of the distinction between signatures and types. In other

$$
\begin{array}{lrl}
\text{IL Type Con's} & \mathrm{A}, \tau ::= & \alpha \mid \tau_1 \to \tau_2 \mid \lambda(\overline{\alpha}).\tau \mid \mathrm{A}(\overline{\tau}) \\
\text{IL Signatures} & \Sigma ::= & [\![= \mathrm{A} : \mathrm{K}]\!] \mid [\![\tau]\!] \mid [\![\overline{\ell : \Sigma}]\!] \mid \Sigma_1 \to \Sigma_2 \mid \\
 & & \forall(\overline{\alpha}).\Sigma \mid \forall(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \mid \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \\
\text{Type Contexts} & \Delta ::= & \emptyset \mid \Delta, \alpha \uparrow \mathrm{K} \mid \Delta, \alpha \downarrow \mathrm{K} \mid \Delta, \alpha \\
\text{Term/Module Contexts} & \Gamma ::= & \emptyset \mid \Gamma, x : \tau \mid \Gamma, \mathrm{X} : \Sigma
\end{array}
$$

Interpretation of type constructors: $\Delta ; \Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{A} : \mathrm{K}$

$$\frac{\Delta ; \Gamma \vdash \mathrm{P} : [\![= \mathrm{A} : \mathrm{K}]\!]}{\Delta ; \Gamma \vdash \mathrm{P} \rightsquigarrow \mathrm{A} : \mathrm{K}} \ (1)$$

Other rules in technical report [4]...

Interpretation of signatures: $\Delta ; \Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma$

$$\frac{}{\Delta ; \Gamma \vdash [\![\mathrm{K}]\!] \rightsquigarrow \exists(\alpha \downarrow \mathrm{K}).[\![= \alpha : \mathrm{K}]\!]} \ (2) \qquad \frac{\Delta ; \Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{A} : \mathrm{K}}{\Delta ; \Gamma \vdash [\![= \mathit{con} : \mathrm{K}]\!] \rightsquigarrow \exists().[\![= \mathrm{A} : \mathrm{K}]\!]} \ (3)$$

$$\frac{\Delta ; \Gamma \vdash \mathit{typ} \rightsquigarrow \tau : \mathbf{T}}{\Delta ; \Gamma \vdash [\![\forall().\mathit{typ}]\!] \rightsquigarrow \exists().[\![\tau]\!]} \ (4) \qquad \frac{\Delta, \overline{\alpha} ; \Gamma \vdash \mathit{typ} \rightsquigarrow \tau : \mathbf{T}}{\Delta ; \Gamma \vdash [\![\forall(\overline{\alpha}).\mathit{typ}]\!] \rightsquigarrow \exists().\forall(\overline{\alpha}).[\![\tau]\!]} \ (5)$$

$$\frac{}{\Delta ; \Gamma \vdash [\![]\!] \rightsquigarrow \exists().[\![]\!]} \ (6) \qquad \frac{\begin{array}{c}\Delta ; \Gamma \vdash \mathit{sig}_1 \rightsquigarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \\ \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1} ; \Gamma, \mathrm{X}_1 : \Sigma_1 \vdash [\![\overline{\ell \triangleright \mathrm{X} : \mathit{sig}}]\!] \rightsquigarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).[\![\overline{\ell : \Sigma}]\!]\end{array}}{\begin{array}{l}\Delta ; \Gamma \vdash [\![\ell_1 \triangleright \mathrm{X}_1 : \mathit{sig}_1, \overline{\ell \triangleright \mathrm{X} : \mathit{sig}}]\!] \\ \qquad \rightsquigarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}, \overline{\alpha \downarrow \mathrm{K}}).[\![\ell_1 : \Sigma_1, \overline{\ell : \Sigma}]\!]\end{array}} \ (7)$$

$$\frac{\Delta ; \Gamma \vdash \mathit{sig}_1 \rightsquigarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \quad \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1} ; \Gamma, \mathrm{X} : \Sigma_1 \vdash \mathit{sig}_2 \rightsquigarrow \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2}{\Delta ; \Gamma \vdash (\mathrm{X} : \mathit{sig}_1) \to \mathit{sig}_2 \rightsquigarrow \exists().\forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \forall().\exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2} \ (8)$$

Fig. 8. Interpretation of signatures

words, polymorphic generalization occurs when a core-level value *val* is encapsulated in an atomic term module [*val*], and polymorphic instantiation happens implicitly when a module path P is used as a core-level expression. This approach deconstructs so-called "let-polymorphism" into its orthogonal component parts. The classic let-polymorphic construct, `let` $x = \mathit{exp}_1$ `in` exp_2, is encodable in our language as "`let` $\mathrm{X} = [\mathit{exp}_1]$ `in` $\{x \mapsto \mathrm{X}\}\mathit{exp}_2$".[2]

Concerning the remaining constructs: Functors are modeled as λ-abstractions. Functor applications restrict the functor and its argument to be paths, but the more general SML-style "P(*mod*)" can be encoded using module-level `let` as "`let` $\mathrm{X} = \mathit{mod}$ `in` $\mathrm{P}(\mathrm{X})$". The two sealing constructs, *mod* `:>` *sig* and *mod* `:` *sig*, model SML's *opaque* and *transparent* signature ascription, respectively.

Figure 8 defines the semantic interpretation of external-language (EL) signatures in terms of internal-language (IL) signatures. The IL is a variant of Dreyer's type system for recursive modules [3], but it is not necessary to be familiar with the whole IL in order to understand how IL signatures are used to interpret EL signatures. The basic idea is that, in IL signatures, abstract type components are modeled as type variables, and their scope is made explicit through the use of universal and existential quantifiers. In fact, IL signatures are very close, both conceptually and formally, to the *semantic objects* employed by the Definition.

[2] We use $\{x \mapsto \mathrm{X}\}$ to denote the capture-avoiding substitution of X for x.

Declarative typing for terms: $\Delta\,;\Gamma \vdash \mathit{exp} : \tau$

$$\frac{\Delta\,;\Gamma \vdash \mathrm{P} : [\![\tau]\!]}{\Delta\,;\Gamma \vdash \mathrm{P} : \tau}\ (9) \qquad \frac{\Delta\,;\Gamma \vdash \mathrm{P} : \forall(\overline{\alpha}).[\![\tau]\!] \quad \Delta \vdash \delta : \overline{\alpha}}{\Delta\,;\Gamma \vdash \mathrm{P} : \delta\tau}\ (10) \qquad \frac{x : \tau \in \Gamma}{\Delta\,;\Gamma \vdash x : \tau}\ (11)$$

$$\frac{\Delta \vdash \tau_1 : \mathbf{T} \quad \Delta\,;\Gamma, x : \tau_1 \vdash \mathit{exp} : \tau_2}{\Delta\,;\Gamma \vdash \lambda x.\mathit{exp} : \tau_1 \to \tau_2}\ (12) \qquad \frac{\Delta\,;\Gamma \vdash \mathit{exp}_1 : \tau_2 \to \tau \quad \Delta\,;\Gamma \vdash \mathit{exp}_2 : \tau_2}{\Delta\,;\Gamma \vdash \mathit{exp}_1(\mathit{exp}_2) : \tau}\ (13)$$

$$\frac{\Delta\,;\Gamma \vdash \mathit{exp} : \tau}{\Delta\,;\Gamma \vdash \mathit{exp} : \mathit{typ} : \tau}\ (14) \qquad \frac{\Delta, \overline{\alpha \uparrow \mathrm{K}}\,;\Gamma \vdash \mathit{mod} : \Sigma \ \mathsf{with}\ \overline{\alpha}\downarrow \quad \Delta, \overline{\alpha \downarrow \mathrm{K}}\,;\Gamma, \mathrm{X} : \Sigma \vdash \mathit{exp} : \tau \quad \overline{\alpha} \# \mathsf{FTV}(\tau)}{\Delta\,;\Gamma \vdash \mathtt{let}\ \mathrm{X} = \mathit{mod}\ \mathtt{in}\ \mathit{exp} : \tau}\ (15)$$

Declarative typing for modules: $\Delta\,;\Gamma \vdash \mathit{mod} : \Sigma \ \mathsf{with}\ \overline{\alpha}\downarrow$

We omit "$\mathsf{with}\ \overline{\alpha}\downarrow$" if $\overline{\alpha} = \emptyset$ (*i.e.,* if *mod* does not define any abstract types).

$$\frac{\mathrm{X} : \Sigma \in \Gamma}{\Delta\,;\Gamma \vdash \mathrm{X} : \Sigma}\ (16) \qquad \frac{\Delta\,;\Gamma \vdash \mathrm{P} : [\![\ldots, \ell : \Sigma, \ldots]\!]}{\Delta\,;\Gamma \vdash \mathrm{P}.\ell : \Sigma}\ (17) \qquad \frac{\Delta\,;\Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{A} : \mathrm{K}}{\Delta\,;\Gamma \vdash [\mathit{con}] : [\![= \mathrm{A} : \mathrm{K}]\!]}\ (18)$$

$$\frac{\Delta\,;\Gamma \vdash \mathit{exp} : \tau}{\Delta\,;\Gamma \vdash [\mathit{exp}] : [\![\tau]\!]}\ (19) \qquad \frac{\Delta, \overline{\alpha}\,;\Gamma \vdash \mathit{val} : \tau}{\Delta\,;\Gamma \vdash [\mathit{val}] : \forall(\overline{\alpha}).[\![\tau]\!]}\ (20) \qquad \frac{}{\Delta\,;\Gamma \vdash [] : [\![]\!]}\ (21)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{mod}_1 : \Sigma_1 \ \mathsf{with}\ \overline{\alpha_1}\downarrow \\ \Delta\,@\,\overline{\alpha_1}\downarrow\,;\Gamma, \mathrm{X}_1 : \Sigma_1 \vdash [\overline{\ell \triangleright \mathrm{X} = \mathit{mod}}] : [\![\overline{\ell : \Sigma}]\!] \ \mathsf{with}\ \overline{\alpha}\downarrow\end{array}}{\Delta\,;\Gamma \vdash [\ell_1 \triangleright \mathrm{X}_1 = \mathit{mod}_1, \overline{\ell \triangleright \mathrm{X} = \mathit{mod}}] : [\![\ell_1 : \Sigma_1, \overline{\ell : \Sigma}]\!] \ \mathsf{with}\ \overline{\alpha_1}, \overline{\alpha}\downarrow}\ (22)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \\ \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}, \overline{\beta}, \overline{\alpha_2 \uparrow \mathrm{K}_2}\,;\Gamma, \mathrm{X} : \Sigma_1 \vdash \mathit{mod} : \Sigma_2 \ \mathsf{with}\ \overline{\alpha_2}\downarrow\end{array}}{\Delta\,;\Gamma \vdash \lambda(\mathrm{X} : \mathit{sig}).\mathit{mod} : \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \forall(\overline{\beta}).\exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2}\ (23)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathrm{P}_1 : \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \forall(\overline{\beta}).\exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2 \quad \Delta\,;\Gamma \vdash \mathrm{P}_2 : \Sigma \\ \Delta \vdash \Sigma \preceq \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \rightsquigarrow \delta_1 \quad \Delta \vdash \delta : \overline{\beta} \quad \Delta \vdash \delta_2 : \overline{\alpha_2 \uparrow \mathrm{K}_2}\end{array}}{\Delta\,;\Gamma \vdash \mathrm{P}_1(\mathrm{P}_2) : \delta\delta_1\delta_2\Sigma_2 \ \mathsf{with}\ \overline{\delta_2\alpha_2}\downarrow}\ (24)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{mod}_1 : \Sigma_1 \ \mathsf{with}\ \overline{\alpha_1}\downarrow \\ \Delta\,@\,\overline{\alpha_1}\downarrow\,;\Gamma, \mathrm{X} : \Sigma_1 \vdash \mathit{mod}_2 : \Sigma_2 \ \mathsf{with}\ \overline{\alpha_2}\downarrow\end{array}}{\Delta\,;\Gamma \vdash \mathtt{let}\ \mathrm{X} = \mathit{mod}_1\ \mathtt{in}\ \mathit{mod}_2 : \Sigma_2 \ \mathsf{with}\ \overline{\alpha_1}, \overline{\alpha_2}\downarrow}\ (25)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \quad \Delta, \overline{\beta \uparrow \mathrm{L}}\,;\Gamma \vdash \mathit{mod} : \Sigma' \ \mathsf{with}\ \overline{\beta}\downarrow \\ \Delta, \overline{\beta \downarrow \mathrm{L}} \vdash \Sigma' \preceq \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \rightsquigarrow \delta' \quad \Delta \vdash \delta : \overline{\alpha \uparrow \mathrm{K}}\end{array}}{\Delta\,;\Gamma \vdash \mathit{mod} :> \mathit{sig} : \delta\Sigma \ \mathsf{with}\ \overline{\delta\alpha}\downarrow}\ (26)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \quad \Delta\,;\Gamma \vdash \mathit{mod} : \Sigma' \ \mathsf{with}\ \overline{\beta}\downarrow \\ \Delta\,@\,\overline{\beta}\downarrow \vdash \Sigma' \preceq \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \rightsquigarrow \delta'\end{array}}{\Delta\,;\Gamma \vdash \mathit{mod} : \mathit{sig} : \delta'\Sigma \ \mathsf{with}\ \overline{\beta}\downarrow}\ (27)$$

Fig. 9. Declarative typing rules for terms and modules

As we explained in Section 2, type variables α that represent abstract type components of modules may be bound in type contexts Δ as *undefined* ($\alpha \uparrow \mathrm{K}$) or as *defined* ($\alpha \downarrow \mathrm{K}$). Type contexts provide an additional binding form (written just α), which is used to represent the implicit type arguments to polymorphic

functions (and generalized functors). This third binding form is necessary because implicit type arguments may be instantiated with types that are either defined or undefined (see Example (c)). All type variables of this third kind are assumed to have base kind $\mathbf{T}$. Term/module contexts, as one would expect, bind term variables x to types τ, and module variables X to signatures Σ. We adopt the convention that contexts are unordered and that commas join together contexts whose domains are disjoint.

The signature interpretation judgment has the form $\Delta\,;\Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma$, which means that $\overline{\alpha}$ (of kinds $\overline{\mathrm{K}}$) represent the opaque type components of EL signature *sig*, and Σ is essentially *sig* with its opaque components defined transparently in terms of $\overline{\alpha}$. For example, **sig type t val v : t end** would be interpreted as $\exists(\alpha \downarrow \mathbf{T}).[\![\mathtt{t} : [\![= \alpha : \mathbf{T}]\!], \mathtt{v} : [\![\alpha]\!]]\!]$.

Most of the rules for interpreting signatures are straightforward. One point of note is in Rule 8 for functor signatures. While the $\forall\rightarrow\exists$ interpretation of SML's generative functors is entirely standard (see Russo's thesis [20]), the "$\forall()$." that precedes the existential in the result signature is unusual. In fact, this is simply a degenerate instance of a *generalized functor signature* (GFS), which may in the general case use the universal quantifier preceding the existential to bind a set of implicit type variables. (For example, see Rule 23, discussed below.)

Figure 9 shows the declarative typing rules for terms and modules. In Section 2 we explained the interpretation of the module typing judgment, and the interpretation of the term typing judgment is the standard one. Before considering the inference rules in detail, let us first define some notation:

We say that A is *defined* in Δ, written $\Delta \vdash \mathrm{A} \downarrow \mathrm{K}$, if $\Delta \vdash \mathrm{A} : \mathrm{K}$ and $\mathsf{FTV}(\mathrm{A}) \subseteq \{\alpha \mid \alpha \downarrow \mathrm{K} \in \Delta\}$. We will write $\Delta \mathbin{@} \overline{\alpha}\downarrow$ to mean $\Delta \backslash \{\alpha \uparrow \mathrm{K} \mid \alpha \in \overline{\alpha}\} \uplus \{\alpha \downarrow \mathrm{K} \mid \alpha \in \overline{\alpha}\}$. We assume and maintain the invariant that all types are kept in β-normal form. (Thus, type substitutions are assumed to implicitly β-normalize.)

Definition 3.1 (Well-Formed Type Substitution).
A type substitution δ mapping Δ to Δ' is *well-formed*, written $\Delta' \vdash \delta : \Delta$, if:

1. $\mathsf{dom}(\delta) \subseteq \mathsf{dom}(\Delta)$
2. $\forall \alpha \uparrow \mathrm{K} \in \Delta.\ \exists \beta \uparrow \mathrm{K} \in \Delta'.\ \beta = \delta\alpha$
3. $\forall \alpha_1 \uparrow \mathrm{K}_1 \in \Delta.\ \forall \alpha_2 \uparrow \mathrm{K}_2 \in \Delta.\ (\delta\alpha_1 = \delta\alpha_2) \Rightarrow (\alpha_1 = \alpha_2)$
4. $\forall \alpha \downarrow \mathrm{K} \in \Delta.\ \Delta' \vdash \delta\alpha \downarrow \mathrm{K}$
5. $\forall \alpha \in \Delta.\ \Delta \vdash \delta\alpha : \mathbf{T}$

Conditions (2) and (4) ensure that substitutions preserve the (un-)definedness of type variables, and condition (3) ensures that undefined variables do not become aliased under substitution. Condition (5) ensures that no restrictions are placed on the types that can be substituted for implicit variables.

Rule 10 performs polymorphic instantiation when coercing a module path P to the term level. Given a path P of *polytype* signature $\forall(\overline{\alpha}).[\![\tau]\!]$, the second premise of the rule nondeterministically guesses a substitution δ for the implicit type arguments $\overline{\alpha}$, which is then applied to the type τ. Rule 20 performs polymorphic generalization when coercing a value *val* to the module level. It acts as a dual to Rule 10 in that it nondeterministically guesses a set of implicit $\overline{\alpha}$ to be added to the context during the typing of *val*.

Rules 21 and 22 define typing for structures. Regarding the latter, there are two points of note. First, all the abstract types defined by the structure (namely, $\overline{\alpha_1}$ and $\overline{\alpha}$) are assumed to be bound as undefined in the initial typing context Δ. Thus, they are in scope throughout the whole structure. Second, note that once the first module binding (mod_1) is typechecked, the remainder of the structure is typechecked in a context where $\overline{\alpha_1}$ are considered defined (namely, $\Delta \mathbin{@} \overline{\alpha_1 \downarrow}$). This ensures that the remainder of the structure will not attempt to redefine $\overline{\alpha_1}$. The typing of module-level `let` (Rule 25) is nearly identical.

Rule 23 defines typing for functors $\lambda(\mathrm{X} : sig).mod$. The second premise adds three sets of type variables to the context when typing the functor body. The $\overline{\alpha_1}$ represent the abstract type components of the functor argument, which are assumed to be defined. The $\overline{\beta}$ represent the implicit type variables over which the functor is polymorphically generalized (in much the same way as the $\overline{\alpha}$ in Rule 20). The $\overline{\alpha_2}$ represent the undefined abstract type components that the functor body *mod* will define itself. Although the choice of $\overline{\alpha_2}$ to add to the context appears to be nondeterministic, the completeness theorem for type inference will show that there is only one way to choose them.

Rule 24 defines typing for functor applications $\mathrm{P}_1(\mathrm{P}_2)$. After checking that P_1 has a valid GFS and that P_2 has some signature Σ, it uses the signature matching judgment $\Delta \vdash \Sigma \preceq \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \rightsquigarrow \delta_1$ to determine whether Σ is coercible to P_1's argument signature. The signature matching judgment, which is defined formally in the technical report [4], returns a substitution δ_1 representing the manifest definitions that Σ provides for the abstract type components $\overline{\alpha_1}$ of P_1's argument signature. This δ_1 has the property that $\Delta \vdash \delta_1 : \overline{\alpha_1 \downarrow \mathrm{K}_1}$. The details of signature matching are largely similar to those in existing accounts of SML.

The fourth premise of Rule 24 nondeterministically guesses a substitution for P_1's implicit type arguments $\overline{\beta}$ (in much the same way as the polymorphic instantiation in Rule 10). Since the functor application will result in the definition of a set of abstract types of the shape specified in P_1's result signature (*i.e.,* in the shape of $\overline{\alpha_2 \downarrow \mathrm{K}_2}$), the last premise of Rule 24 requires that such a set of abstract types already exist, undefined, in the context Δ. These abstract types are denoted by $\overline{\delta_2 \alpha_2}$.

Finally, Rules 26 and 27 define typing for opaque and transparent sealing, respectively. In both rules, the signature Σ' of the module *mod* is matched against the interpretation $\exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma$ of the ascribed signature *sig*. This results in a substitution δ', which conveys how *mod* implements the abstract type components $\overline{\alpha}$ of *sig*. In the case of opaque sealing, this information is irrelevant, since the signature of the sealed module keeps the $\overline{\alpha}$ abstract (albeit renamed by δ, whose role is similar to that of δ_2 in Rule 24). In the case of transparent sealing, the substitution δ' obtained from signature matching is applied to Σ in the signature of the sealed module, thus allowing *mod*'s definitions for the $\overline{\alpha}$ to leak out.

In both sealing rules, *mod* is permitted to define a set of abstract types $\overline{\beta}$. However, Rule 26 adds $\overline{\beta}$ to the context Δ when typechecking *mod*, whereas Rule 27 assumes $\overline{\beta}$ are already present in Δ. The reason for this is as follows. If *mod* is opaquely sealed, then $\overline{\beta}$ cannot escape the scope of the sealed module—

Signature inference for modules: $\Delta\,;\Gamma \vdash \mathit{mod} \Rightarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).(\Sigma;\theta)$

We omit "$\exists(\overline{\alpha \downarrow \mathrm{K}})$." if $\overline{\alpha \downarrow \mathrm{K}} = \emptyset$ (*i.e.,* if *mod* does not define any abstract types).

$$\frac{\Delta\,;\Gamma \vdash \mathrm{P} : \Sigma}{\Delta\,;\Gamma \vdash \mathrm{P} \Rightarrow (\Sigma;\mathsf{id})} \quad (28) \qquad \frac{\Delta\,;\Gamma \vdash \mathit{con} \rightsquigarrow \mathrm{A} : \mathrm{K}}{\Delta\,;\Gamma \vdash [\mathit{con}] \Rightarrow ([\![= \mathrm{A} : \mathrm{K}]\!];\mathsf{id})} \quad (29) \qquad \frac{}{\Delta\,;\Gamma \vdash [] \Rightarrow ([\![]\!];\mathsf{id})} \quad (30)$$

$$\frac{\Delta\,;\Gamma \vdash \mathit{exp} \Rightarrow (\tau;\theta) \quad \mathit{exp} \text{ not a } \mathit{val}}{\Delta\,;\Gamma \vdash [\mathit{exp}] \Rightarrow ([\![\tau]\!];\theta)} \quad (31) \qquad \frac{\Delta\,;\Gamma \vdash \mathit{val} \Rightarrow (\tau;\theta) \quad \overline{\boldsymbol{\alpha}} = \mathsf{UV}(\tau)\backslash\mathsf{UV}(\theta\Gamma)}{\Delta\,;\Gamma \vdash [\mathit{val}] \Rightarrow (\forall(\overline{\boldsymbol{\alpha}}).[\![\tau]\!];\theta)} \quad (32)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{mod}_1 \Rightarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\Sigma_1;\theta_1) \\ \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}\,;\theta_1\Gamma, \mathrm{X}_1 : \Sigma_1 \vdash [\overline{\ell \triangleright \mathrm{X} = \mathit{mod}}] \Rightarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).([\![\overline{\ell : \Sigma}]\!];\theta_2)\end{array}}{\Delta\,;\Gamma \vdash [\ell_1 \triangleright \mathrm{X}_1 = \mathit{mod}_1, \overline{\ell \triangleright \mathrm{X} = \mathit{mod}}] \Rightarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}, \overline{\alpha \downarrow \mathrm{K}}).([\![\ell_1 : \theta_2\Sigma_1, \overline{\ell : \Sigma}]\!];\theta_2\theta_1|_\Gamma)} \quad (33)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \quad \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}\,;\Gamma, \mathrm{X} : \Sigma_1 \vdash \mathit{mod} \Rightarrow \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).(\Sigma_2;\theta) \\ \overline{\boldsymbol{\alpha}} = \mathsf{UV}(\Sigma_2)\backslash\mathsf{UV}(\theta\Gamma) \quad \overline{\alpha_1}, \overline{\alpha_2} \# \mathsf{FTV}(\theta)\end{array}}{\Delta\,;\Gamma \vdash \lambda(\mathrm{X} : \mathit{sig}).\mathit{mod} \Rightarrow (\forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \forall(\overline{\boldsymbol{\alpha}}).\exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2;\theta)} \quad (34)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathrm{P}_1 : \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \forall(\overline{\beta}).\exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2 \quad \Delta\,;\Gamma \vdash \mathrm{P}_2 : \Sigma \\ \Delta \vdash \Sigma \preceq \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \Rightarrow (\delta;\theta) \quad \overline{\boldsymbol{\alpha}}\ \mathsf{fresh}\end{array}}{\Delta\,;\Gamma \vdash \mathrm{P}_1(\mathrm{P}_2) \Rightarrow \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).(\{\overline{\beta \mapsto \boldsymbol{\alpha}}\}\theta\delta\Sigma_2;\theta)} \quad (35)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{mod}_1 \Rightarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}).(\Sigma_1;\theta_1) \\ \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}\,;\theta_1\Gamma, \mathrm{X} : \Sigma_1 \vdash \mathit{mod}_2 \Rightarrow \exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).(\Sigma_2;\theta_2)\end{array}}{\Delta\,;\Gamma \vdash \mathtt{let}\ \mathrm{X} = \mathit{mod}_1\ \mathtt{in}\ \mathit{mod}_2 \Rightarrow \exists(\overline{\alpha_1 \downarrow \mathrm{K}_1}, \overline{\alpha_2 \downarrow \mathrm{K}_2}).(\Sigma_2;\theta_2\theta_1|_\Gamma)} \quad (36)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \quad \Delta\,;\Gamma \vdash \mathit{mod} \Rightarrow \exists(\overline{\beta \downarrow \mathrm{L}}).(\Sigma_1;\theta_1) \\ \Delta, \overline{\beta \downarrow \mathrm{L}} \vdash \Sigma_1 \preceq \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \Rightarrow (\delta;\theta_2) \quad \overline{\beta} \# \mathsf{FTV}(\theta_2\theta_1|_\Gamma)\end{array}}{\Delta\,;\Gamma \vdash \mathit{mod} :\!> \mathit{sig} \Rightarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).(\Sigma;\theta_2\theta_1|_\Gamma)} \quad (37)$$

$$\frac{\begin{array}{c}\Delta\,;\Gamma \vdash \mathit{sig} \rightsquigarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \quad \Delta\,;\Gamma \vdash \mathit{mod} \Rightarrow \exists(\overline{\beta \downarrow \mathrm{L}}).(\Sigma_1;\theta_1) \\ \Delta, \overline{\beta \downarrow \mathrm{L}} \vdash \Sigma_1 \preceq \exists(\overline{\alpha \downarrow \mathrm{K}}).\Sigma \Rightarrow (\delta;\theta_2)\end{array}}{\Delta\,;\Gamma \vdash \mathit{mod} : \mathit{sig} \Rightarrow \exists(\overline{\beta \downarrow \mathrm{L}}).(\delta\Sigma;\theta_2\theta_1|_\Gamma)} \quad (38)$$

Fig. 10. Inference rules for modules

i.e., $\overline{\beta}$ are *local* abstract types, which are thus introduced into scope locally by Rule 26. If *mod* is transparently sealed, then $\overline{\beta}$ can leak out into the signature of the sealed module, and must therefore be bound in the surrounding context Δ.

4 Type Inference Algorithm

The type inference algorithm for our language is based closely on Algorithm $\mathcal{W}$. We employ unification variables (u-vars), written $\boldsymbol{\alpha}$, in the usual way. In particular, we do not explicitly bind u-vars in the context Δ. The u-vars appearing free in an expression E, which we write as $\mathsf{UV}(E)$, are all taken to have kind $\mathbf{T}$.

The inference judgment for terms has the familiar form $\Delta\,;\Gamma \vdash \mathit{exp} \Rightarrow (\tau;\theta)$. Here, Δ, Γ, and *exp* are considered inputs. τ is the *principal type scheme* of *exp*, meaning that any other type that one can assign to *exp* declaratively must be a u-var substitution instance of τ. Lastly, θ is an idempotent u-var substitution

Signature subsumption: $\Delta \vdash \Sigma_1 \sqsubseteq \Sigma_2$

$$\frac{}{\Delta \vdash [\![= \mathrm{A} : \mathrm{K}]\!] \sqsubseteq [\![= \mathrm{A} : \mathrm{K}]\!]} \ (39) \quad \frac{\Delta\,; \mathrm{X} : \Sigma \vdash \mathrm{X} : \tau}{\Delta \vdash \Sigma \sqsubseteq [\![\tau]\!]} \ (40) \quad \frac{\Delta, \overline{\alpha}\,; \mathrm{X} : \Sigma \vdash \mathrm{X} : \tau}{\Delta \vdash \Sigma \sqsubseteq \forall(\overline{\alpha}).[\![\tau]\!]} \ (41)$$

$$\frac{}{\Delta \vdash [\![]\!] \sqsubseteq [\![]\!]} \ (42) \quad \frac{\Delta \vdash \Sigma_1 \sqsubseteq \Sigma_1' \quad \Delta \vdash [\![\overline{\ell : \Sigma}]\!] \sqsubseteq [\![\overline{\ell : \Sigma'}]\!]}{\Delta \vdash [\![\ell_1 : \Sigma_1, \overline{\ell : \Sigma}]\!] \sqsubseteq [\![\ell_1 : \Sigma_1', \overline{\ell : \Sigma'}]\!]} \ (43)$$

$$\frac{\Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}, \overline{\beta'}, \overline{\alpha_2 \downarrow \mathrm{K}_2} \vdash \delta : \overline{\beta} \quad \Delta, \overline{\alpha_1 \downarrow \mathrm{K}_1}, \overline{\beta'}, \overline{\alpha_2 \downarrow \mathrm{K}_2} \vdash \delta\Sigma_2 \sqsubseteq \Sigma_2'}{\Delta \vdash \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \forall(\overline{\beta}).\exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2 \sqsubseteq \forall(\overline{\alpha_1 \downarrow \mathrm{K}_1}).\Sigma_1 \to \forall(\overline{\beta'}).\exists(\overline{\alpha_2 \downarrow \mathrm{K}_2}).\Sigma_2'} \ (44)$$

Fig. 11. Signature subsumption

whose domain is a subset of $\mathsf{UV}(\Gamma)$. (In some rules, this is enforced by explicitly writing $\theta|_\Gamma$, which denotes θ with its domain restricted to $\mathsf{UV}(\Gamma)$.) It represents the minimal substitution that must be applied to Γ in order to make *exp* well-typed. The rules for this judgment are standard (see the technical report [4]).

Figure 10 defines the inference judgment for modules, which has the form $\Delta\,; \Gamma \vdash \mathit{mod} \Rightarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).(\Sigma; \theta)$. Here, Δ, Γ, and *mod* are considered inputs. The $\overline{\alpha \downarrow \mathrm{K}}$ represent the abstract types that *mod* wants to define. Unlike the declarative judgment, inference does not make any assumption that $\overline{\alpha}$ are bound (as undefined) in the input context Δ.

Σ is the *principal signature scheme* of *mod*, meaning that any other signature that one can assign to *mod* declaratively must be "less general" than some u-var substitution instance of Σ. In traditional presentations of HM, "less general" is characterized by means of a *subsumption* relation on polytypes. Since polytypes in our language are just a special case of signatures, we generalize subsumption to be a relation on signatures. Defined in Figure 11, the judgment $\Delta \vdash \Sigma_1 \sqsubseteq \Sigma_2$ says that Σ_1 is more general than Σ_2. Note that Rules 40 and 41 exploit the polymorphic instantiation offered by the declarative Rules 9 and 10.

As in the inference judgment for terms, θ is the minimal substitution to be applied to Γ in order to make *mod* well-typed. An important point is that the free variables of θ may include the abstract types $\overline{\alpha}$ defined by *mod*. This is critical because it enables forward references to abstract types. For example, suppose that, as a result of inference for an earlier binding in the program, a variable X is bound in Γ with $[\![\boldsymbol{\beta} \to \boldsymbol{\beta}]\!]$. If during inference for *mod* the u-var $\boldsymbol{\beta}$ is unified with one of the $\overline{\alpha}$ defined by *mod*, then that constitutes a forward reference from the signature of X to an abstract type defined later in the program, and we want it in general to be accepted (for the reasons explained in Section 2).

That said, there are instances in which forward references must be prohibited in order to ensure soundness of type inference. One such instance is the inference rule for functors (Rule 34), which includes a side condition stipulating that the abstract type components of the argument and result ($\overline{\alpha_1}$ and $\overline{\alpha_2}$, respectively) do not appear in the free variables of the output substitution θ. This restriction is necessitated by the fact that $\overline{\alpha_1}$ and $\overline{\alpha_2}$ are *local* abstract types that are only in scope within the body of the functor. Indeed, the declarative rule for functors (Rule 23) imposes the same restriction—when typechecking the functor body it

adds $\overline{\alpha_1}$ and $\overline{\alpha_2}$ to Δ instead of assuming that they were already bound in it to begin with. This has the effect that the typing of earlier bindings in the program cannot make forward references to $\overline{\alpha_1}$ and $\overline{\alpha_2}$.

We have verified manually that the type inference algorithm is sound and complete with respect to the declarative semantics. Here we state the soundness and completeness theorems in abbreviated form (only giving the cases concerning modules, and with some of the side conditions elided). The full theorem statements, together with relevant auxiliary judgments, are given in [4].

Theorem 4.1 (Soundness).
Assuming certain side conditions on Γ and mod,
if $\Delta\,;\Gamma \vdash mod \Rightarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).(\Sigma;\theta)$, then $\Delta, \overline{\alpha \uparrow \mathrm{K}}\,;\theta\Gamma \vdash mod : \Sigma$ with $\overline{\alpha}\downarrow$.

Theorem 4.2 (Completeness).
Assuming certain side conditions on Γ, Γ', θ, and mod,
if $\Delta' \supseteq \Delta, \overline{\alpha \uparrow \mathrm{K}}$ and $\Delta' \vdash \theta\Gamma \sqsubseteq \Gamma'$ and $\Delta'\,;\Gamma' \vdash mod : \Sigma$ with $\overline{\alpha}\downarrow$,
then $\Delta\,;\Gamma \vdash mod \Rightarrow \exists(\overline{\alpha \downarrow \mathrm{K}}).(\Sigma';\theta')$
and there exists θ'' such that $\theta''\theta'\Gamma = \theta\Gamma$ and $\Delta' \vdash \theta''\Sigma' \sqsubseteq \Sigma$.

The premise $\Delta' \vdash \theta\Gamma \sqsubseteq \Gamma'$ in the completeness statement refers to the natural generalization of signature subsumption to context subsumption. We use it here to build a weakening property of declarative derivations directly into the induction hypothesis, so as to avoid having to prove it separately.

5 Related and Future Work

Russo describes a type inference algorithm for ML with higher-order and first-class modules, in which he uses alternating $\exists\forall$-quantification to track the scoping restrictions on abstract types imposed by The Definition [20]. This technique is also known as *unification under a mixed prefix* [14]. Although Russo states soundness and completeness conjectures, he does not attempt to prove them, and the implementation of his algorithm in Moscow ML rejects Example (a).

Many researchers have investigated the problem of type inference for a wide spectrum of languages in the design space between Hindley-Milner and System F. For example, Odersky and Läufer consider the problem of type inference in the presence of abstract data types and higher-order polymorphism [18]. Their type system relies on programmer-provided type annotations for handling polymorphic function arguments and existentials. It does not, however, include explicit type abstractions, and thus cannot directly model ML functors. Due to the presence of programmer-declared existential types, their inference algorithm, like Russo's, has to perform unification under a mixed prefix.

For future work, we are interested in extending our type system and its inference algorithm to more complete languages, in particular to full SML, as well as to languages with *applicative functors* [9]. The most prominent example of such a language is OCaml. Currently, the OCaml compiler rejects all four of our examples with error messages similar to TILT's. This comes as no surprise since, applicative functors aside, the typecheckers of both compilers are based closely on the Harper-Lillibridge/Leroy type system [6, 10].

The problems concerning type inference and modules that we have explored in this work were originally discovered during the development of a modular account of Haskell-style type classes in ML [5]. Therefore, we hope to be able to adapt the techniques developed in this paper in order to obtain a similar soundness and completeness result for modular type classes.

References

1. Luis Damas and Robin Milner. Principal type schemes for functional programs. In *POPL '82*.
2. Derek Dreyer. Recursive type generativity. To appear in Journal of Functional Programming. Original version appeared in *ICFP '05*.
3. Derek Dreyer. Practical type theory for recursive modules. Technical Report TR-2006-07, University of Chicago, Department of Computer Science, August 2006.
4. Derek Dreyer and Matthias Blume. Principal type schemes for modular programs. Technical Report TR-2007-02, Univ. of Chicago Comp. Sci. Dept., January 2007.
5. Derek Dreyer, Robert Harper, and Manuel M. T. Chakravarty. Modular type classes. In *POPL '07*.
6. Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94*.
7. Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.
8. Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.
9. Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL 95*.
10. Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL '94*.
11. Xavier Leroy. *Polymorphic Typing of an Algorithmic Language*. PhD thesis, Université Paris 7, 1992.
12. Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, May 1997.
13. David MacQueen, 2006. Private communication.
14. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.
15. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–75, 1978.
16. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
17. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.
18. Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *POPL '96*, pages 54–67.
19. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
20. Claudio V. Russo. *Types for Modules*. PhD thesis, University of Edinburgh, 1998.
21. Claudio V. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4):348–374, 2000.
22. Andrew K. Wright. Polymorphic references for mere mortals. In *ESOP '92*.