COVER PAGE

CONFERENCE SUBMISSION

The 1999 International Conference on Principles and Practice of Declarative Programming (PPDP'99)

Authors: Dr. Claudio V. Russo
Corresponding Author: Dr. Claudio V. Russo
Postal Address:
3F1 12 Dryden St.
Edinburgh, EH7 4PN
Scotland, UK
Email: `cvr@dcs.ed.ac.uk`
Phone:
- +44 (131) 554 4848 (home)
- +44 (131) 650 5163 (work)

Fax: +44 (131) 667 7209
Institute:
LFCS, Division of Informatics, University of Edinburgh,
JCMB, King's Buildings, Mayfield Road,
Edinburgh EH9 3JZ, UK

Title: Non-Dependent Types for Standard ML Modules

Abstract: Two of the distinguishing features of Standard ML Modules are its term dependent type syntax and the use of type generativity in its static semantics. From a type-theoretic perspective, the former suggests that the language involves first-order dependent types, while the latter has been regarded as an extra-logical device that bears no direct relation to type-theoretic constructs. We reformulate the existing semantics of Modules to reveal a purely second-order type theory. In particular, we show that generativity corresponds precisely to existential quantification over types and that the remainder of the Modules type structure is based exclusively on the second-order notions of type parameterisation, universal type quantification and subtyping. Our account is more direct than others and has been shown to scale naturally to both higher-order and first-class modules.

Keywords: dependent types, generativity, language design, modules, Standard ML, type theory, existential types

Pages: This page + 1 blank page + 16 pages ALL meant for the proceedings.

COVER PAGE

THIS PAGE INTENTIONALLY LEFT BLANK

Non-Dependent Types for Standard ML Modules

Claudio V. Russo

LFCS, Division of Informatics, University of Edinburgh,
JCMB, KB, Mayfield Road, Edinburgh EH9 3JZ
www: `http://www.dcs.ed.ac.uk/~cvr`, email: `cvr@dcs.ed.ac.uk`
(funded under a grant from the EPSRC)

Abstract. Two of the distinguishing features of Standard ML Modules are its term dependent type syntax and the use of type generativity in its static semantics. From a type-theoretic perspective, the former suggests that the language involves first-order dependent types, while the latter has been regarded as an extra-logical device that bears no direct relation to type-theoretic constructs. We reformulate the existing semantics of Modules to reveal a purely second-order type theory. In particular, we show that generativity corresponds precisely to existential quantification over types and that the remainder of the Modules type structure is based exclusively on the second-order notions of type parameterisation, universal type quantification and subtyping. Our account is more direct than others and has been shown to scale naturally to both higher-order and first-class modules.

1 Introduction

Standard ML [11] has a rich and influential Modules language. Core language definitions of type and term identifiers can be packaged together into possibly nested terms called *structures*. Access to structure components is by the dot notation and provides good control of the name space in a large program.

The use of the dot notation to project types from terms suggests that the type structure of Standard ML is based on first-order dependent types. In this interpretation, proposed in [9] and refined in [2], nested structures are modelled as dependent pairs whose types are first-order existentially quantified types. Standard ML functors, that define functions mapping structures to structures, are modelled using dependent functions whose types are first-order universally quantified types. Adopting first-order dependent types in a programming language is problematic as it rules out the consistent extension to first-class modules [2] and violates the phase distinction between compile-time type checking and run-time evaluation [3]. More recently proposed module calculi [1, 4–8] that capture some, but not all, of the features of Standard ML, and significantly generalise them, resort to non-standard formulations of dependent types. Though appealing, these calculi have undesirable properties (undecidable subtyping in [1, 8] and the lack of principal types in [1, 4–6, 8]).

Core Types	u ::=	t	type identifier
	|	$\mathrm{u} \to \mathrm{u}'$	function type
	|	**int**	integers
	|	sp.t	type projection
Signature Bodies	B ::=	**type** t = u; B	transparent type specification
	|	**type** t; B	opaque type specification
	|	**val** x : u; B	value specification
	|	**structure** X : S; B	structure specification
	|	ϵ_{B}	empty body
Signature Expressions	S ::=	**sig** B **end**	encapsulated body

Fig. 1. Type Syntax of Mini-SML

In this paper, we take a second look at the type structure of Standard ML Modules by studying a representative toy language, Mini-SML. The static semantics of Mini-SML is based directly on that of Standard ML, but our choice of notation reveals an underlying type structure that, despite the term dependent type syntax, is based entirely on the simpler, second-order notions of type parameterisation, universal type quantification and subtyping. What remains to be explained is the role of type generativity in the semantics, that lends it a procedural, non type-theoretic flavour by requiring a global state of generated types to be maintained and updated during type checking. We explain and eliminate generativity by presenting an alternative, but equivalent, static semantics based on the introduction and elimination of second-oder existential types, thus accounting for all of Mini-SML's type structure in a purely second-order type theory.

2 Syntax

Mini-SML includes the essential features of Standard ML Modules but, for presentation reasons, is constructed on top of a simple Core language of explicitly typed, monomorphic functions. The author's thesis [12], on which this paper is based, presents similar results for a generic Core language that encompasses ones like Standard ML's (which supports the definition of parameterised types, is implicitly typed, and polymorphic). The *type* and *term* syntax of Mini-SML is defined by the grammar in Figures 1 and 2, where $\mathrm{t} \in \mathrm{TypId}$, $\mathrm{x} \in \mathrm{ValId}$, $\mathrm{X} \in \mathrm{StrId}$, and $\mathrm{F} \in \mathrm{FunId}$ range over disjoint sets of type, value, structure and functor identifiers.

A *core type* u may be used to define a type identifier or to specify the type of a Core value. These are just the types of a simple functional language, extended with the projection sp.t of a type component from a structure path. A *signature body* B is a sequential specification of a structure's components. A type component may be specified *transparently*, by equating it with a type, or *opaquely*, permitting a variety of *realisations*. Value and structure components are specified by their type and signature. The specifications in a body are dependent in that subsequent specifications may refer to previous ones. A *signature expression*

Core Expressions	$\mathrm{e} ::=$	x	value identifier
	$\mid$	$\lambda \mathrm{x} : \mathrm{u}.\mathrm{e}$	function
	$\mid$	$\mathrm{e}\ \mathrm{e}'$	application
	$\mid$	i	integer constant
	$\mid$	$\mathrm{sp}.\mathrm{x}$	value projection
Structure Paths	$\mathrm{sp} ::=$	X	structure identifier
	$\mid$	$\mathrm{sp}.\mathrm{X}$	structure projection
Structure Bodies	$\mathrm{b} ::=$	$\mathbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{b}$	type definition
	$\mid$	$\mathbf{datatype}\ \mathrm{t}\ =\ \mathrm{u}\ \mathbf{with}\ \mathrm{x}, \mathrm{x}'; \mathrm{b}$	datatype definition
	$\mid$	$\mathbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b}$	value definition
	$\mid$	$\mathbf{structure}\ \mathrm{X}\ =\ \mathrm{s}; \mathrm{b}$	structure definition
	$\mid$	$\mathbf{local}\ \mathrm{X}\ =\ \mathrm{s}\ \mathbf{in}\ \mathrm{b}$	local structure definition
	$\mid$	$\mathbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \mathbf{in}\ \mathrm{b}$	functor definition
	$\mid$	ϵ_{b}	empty body
Structure Expressions	$\mathrm{s} ::=$	sp	structure path
	$\mid$	$\mathbf{struct}\ \mathrm{b}\ \mathbf{end}$	structure body
	$\mid$	$\mathrm{F}(\mathrm{s})$	functor application
	$\mid$	$\mathrm{s} : \mathrm{S}$	transparent constraint
	$\mid$	$\mathrm{s} :> \mathrm{S}$	opaque constraint

Fig. 2. Term Syntax of Mini-SML

S merely encapsulates a body. A structure *matches* a signature expression if it provides an implementation for all of the specified components, and possibly more.

Core expressions e describe a simple functional language extended with the projection of a value identifier from a structure path. A *structure path* sp is a reference to a bound structure identifier or the projection of one of its substructures. A *structure body* b is a sequence of definitions: subsequent definitions in the body may refer to previous ones. A type definition abbreviates a type. A datatype definition generates a new (recursive) type with value *constructor* x and value *destructor* x'. Value, structure and local definitions bind term identifiers to the values of expressions. A functor definition introduces a named function on structures: X is the functor's formal argument, S specifies the argument's type, and s is the functor's body that may refer to X. The functor may be applied to any argument that matches S. A *structure expression* s evaluates to a structure. It may be a path or an encapsulated structure body, whose type, value and structure definitions become the components of the structure. The application of a functor evaluates its body with respect to the value of the actual argument, generating any new types created by the body. A transparent constraint restricts the visibility of the structure's components to those specified in the signature, which the structure must match, but preserves the realisations of type components with opaque specifications. An opaque constraint is similar, but generates new, and thus *abstract*, types for type components with opaque specifications.

Standard ML only permits functor definitions in the top-level syntax. Mini-SML allows local functor definitions in structure bodies, which can now serve as the top-level: this generalisation avoids the need for a separate top-level syntax.

$$
\begin{array}{rll}
\alpha \in \mathit{Var} & \stackrel{\text{def}}{=} \{\alpha, \beta, \delta, \gamma, \ldots\} & \text{type variables} \\
M, N, P, Q \in \mathit{VarSet} & \stackrel{\text{def}}{=} \mathrm{Fin}(\mathit{Var}) & \text{sets of type variables} \\
u \in \mathit{Type} & ::= \alpha & \text{type variable} \\
& \mid\ u \to u' & \text{function space} \\
& \mid\ \mathrm{int} & \text{integers} \\
\varphi \in \mathit{Real} & \stackrel{\text{def}}{=} \mathit{Var} \stackrel{\text{fin}}{\to} \mathit{Type} & \text{realisations} \\
\mathcal{S} \in \mathit{Str} & \stackrel{\text{def}}{=} \left\{ \begin{array}{l|l} \mathcal{S}_{\mathrm{t}} \cup & \mathcal{S}_{\mathrm{t}} \in \mathrm{TypId} \stackrel{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{S}_{\mathrm{x}} \cup & \mathcal{S}_{\mathrm{x}} \in \mathrm{ValId} \stackrel{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{S}_{\mathrm{X}} & \mathcal{S}_{\mathrm{X}} \in \mathrm{StrId} \stackrel{\text{fin}}{\to} \mathit{Str} \end{array} \right\} & \text{semantic structures} \\
\mathcal{L} \in \mathit{Sig} & ::= \Lambda P.\mathcal{S} & \text{semantic signatures} \\
\mathcal{X} \in \mathit{ExStr} & ::= \exists P.\mathcal{S} & \text{existential structures} \\
\mathcal{F} \in \mathit{Fun} & ::= \forall P.\mathcal{S} \to \mathcal{X} & \text{semantic functors} \\
\mathcal{C} \in \mathit{Context} & \stackrel{\text{def}}{=} \left\{ \begin{array}{l|l} \mathcal{C}_{\mathrm{t}} \cup & \mathcal{C}_{\mathrm{t}} \in \mathrm{TypId} \stackrel{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{C}_{\mathrm{x}} \cup & \mathcal{C}_{\mathrm{x}} \in \mathrm{ValId} \stackrel{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{C}_{\mathrm{X}} \cup & \mathcal{C}_{\mathrm{X}} \in \mathrm{StrId} \stackrel{\text{fin}}{\to} \mathit{Str}, \\ \mathcal{C}_{\mathrm{F}} & \mathcal{C}_{\mathrm{F}} \in \mathrm{FunId} \stackrel{\text{fin}}{\to} \mathit{Fun} \end{array} \right\} & \text{semantic contexts}
\end{array}
$$

Fig. 3. Semantic Objects of Mini-SML

3 Semantic Objects

Following Standard ML [11], the static semantics of Mini-SML distinguishes between the syntactic types of the language and their semantic counterparts called *semantic objects*. Semantic objects play the role of types in the static semantics. Figure 3 defines the semantic objects of Mini-SML. We let $\mathcal{O}$ range over all semantic objects.

Notation: For sets A and B, $\mathrm{Fin}(A)$ denotes the set of *finite subsets* of A, and $A \stackrel{\text{fin}}{\to} B$ denotes the set of *finite maps* from A to B. Let f and g be finite maps. $\mathcal{D}(f)$ denotes the *domain of definition* of f. The finite map $f + g$ has domain $\mathcal{D}(f) \cup \mathcal{D}(g)$ and values $(f + g)(a) \stackrel{\text{def}}{=}$ if $a \in \mathcal{D}(g)$ then $g(a)$ else $f(a)$.

Type variables $\alpha \in \mathit{Var}$ are just variables ranging over *semantic types* $u \in \mathit{Type}$. The latter are the semantic counterparts of syntactic core types, and are used to record the denotations of type identifiers and the types of value identifiers.

A *realisation* $\varphi \in \mathit{Real}$ maps type variables to semantic types and defines a *substitution* on type variables in the usual way. The operation of applying a realisation φ to an object $\mathcal{O}$ is written $\varphi(\mathcal{O})$.

Semantic structures $\mathcal{S} \in \mathit{Str}$ are used as the types of structure identifiers and paths. A semantic structure maps type components to the types they denote, and value and structure components to the types they inhabit. For clarity, we define the extension functions $\mathrm{t} \rhd u, \mathcal{S} \stackrel{\text{def}}{=} \{\mathrm{t} \mapsto u\} + \mathcal{S}$, $\mathrm{x} : u, \mathcal{S} \stackrel{\text{def}}{=} \{\mathrm{x} \mapsto u\} + \mathcal{S}$, and $\mathrm{X} : \mathcal{S}, \mathcal{S}' \stackrel{\text{def}}{=} \{\mathrm{X} \mapsto \mathcal{S}\} + \mathcal{S}'$, and let $\epsilon_{\mathcal{S}}$ denote the empty structure $\emptyset$.

Note that Λ, $\exists$ and $\forall$ bind finite sets of type variables.

A *semantic signature* $\Lambda P.\mathcal{S}$ is a parameterised type: it describes the family of structures $\varphi(\mathcal{S})$, for φ a realisation of the parameters in P.

The *existential structure* $\exists P.\mathcal{S}$, on the other hand, is a quantified type: variables in P are existentially quantified in $\mathcal{S}$ and thus abstract.

A *semantic functor* $\forall P.\mathcal{S} \to \mathcal{X}$ describes the type of a functor identifier: the universally quantified variables in P are bound simultaneously in the functor's domain, $\mathcal{S}$, and its range, $\mathcal{X}$. These variables capture the type components of the domain on which the functor behaves polymorphically; their possible occurrence in the range caters for the propagation of type identities from the functor's actual argument: functors are polymorphic functions on structures. The range $\mathcal{X}$ of a functor is an existential structure $\mathcal{X} \equiv \exists Q.\mathcal{S}'$. Q is the functor's set of generative type variables, as described in the Definition of Standard ML [11]. When a functor with this range is applied, the type of the result is a variant of $\mathcal{S}'$, obtained by replacing variables in Q with new, generative variables.

The Definition of Standard ML [11] is decidedly non-committal in its choice of binding operators, using the uniform notation of parenthesised variable sets to indicate binding in semantic objects. We prefer to differentiate binders with the more suggestive notation Λ, $\forall$ and $\exists$.

A *context* $\mathcal{C}$ is finite map mapping type identifiers to the semantic types they denote, and value, structure and functor identifiers to the types they inhabit. For clarity, we define the extension functions $\mathcal{C}, \mathrm{t} \rhd u \stackrel{\text{def}}{=} \mathcal{C} + \{\mathrm{t} \mapsto u\}$, $\mathcal{C}, \mathrm{x} : u \stackrel{\text{def}}{=} \mathcal{C} + \{\mathrm{x} \mapsto u\}$, $\mathcal{C}, \mathrm{X} : \mathcal{S} \stackrel{\text{def}}{=} \mathcal{C} + \{\mathrm{x} \mapsto \mathcal{S}\}$, and $\mathcal{C}, \mathrm{F} : \mathcal{F} \stackrel{\text{def}}{=} \mathcal{C} + \{\mathrm{F} \mapsto \mathcal{F}\}$.

We let $\mathcal{V}(\mathcal{O})$ denote the set of variables occurring *free* in $\mathcal{O}$, where the notions of free and bound variable are defined as usual. Furthermore, we *identify* semantic objects that differ only in a renaming of bound type variables (α-conversion).

The operation of applying a realisation to a type (substitution) is extended to all semantic objects in the usual way, taking care to avoid the capture of free variables by bound variables.

Definition 1 (Enrichment Relation) *Given two structures $\mathcal{S}$ and $\mathcal{S}'$, $\mathcal{S}$* enriches *$\mathcal{S}'$, written $\mathcal{S} \succeq \mathcal{S}'$, if and only if*

- $\mathcal{D}(\mathcal{S}) \supseteq \mathcal{D}(\mathcal{S}')$,
- *for all* $\mathrm{t} \in \mathcal{D}(\mathcal{S}')$, $\mathcal{S}(\mathrm{t}) = \mathcal{S}'(\mathrm{t})$,
- *for all* $\mathrm{x} \in \mathcal{D}(\mathcal{S}')$, $\mathcal{S}(\mathrm{x}) = \mathcal{S}'(\mathrm{x})$, *and*
- *for all* $\mathrm{X} \in \mathcal{D}(\mathcal{S}')$, $\mathcal{S}(\mathrm{X}) \succeq \mathcal{S}'(\mathrm{X})$.

Enrichment is a pre-order that defines a *subtyping* relation on semantic structures (i.e. $\mathcal{S}$ is a subtype of $\mathcal{S}'$ if and only if $\mathcal{S} \succeq \mathcal{S}'$).

Definition 2 (Functor Instantiation) *A semantic functor $\forall P.\mathcal{S} \to \mathcal{X}$ instantiates to a* functor instance *$\mathcal{S}' \to \mathcal{X}'$, written $\forall P.\mathcal{S} \to \mathcal{X} > \mathcal{S}' \to \mathcal{X}'$, if and only if $\varphi(\mathcal{S}) = \mathcal{S}'$ and $\varphi(\mathcal{X}) = \mathcal{X}'$, for some realisation φ with $\mathcal{D}(\varphi) = P$.*

Definition 3 (Signature Matching) *A semantic structure $\mathcal{S}$* matches *a signature $\Lambda P.\mathcal{S}'$ if and only if there exists a realisation φ with $\mathcal{D}(\varphi) = P$ such that $\mathcal{S} \succeq \varphi(\mathcal{S}')$.*

$$\boxed{\mathcal{C} \vdash \mathrm{u} \rhd u} \qquad \frac{\mathrm{t} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{t} \rhd \mathcal{C}(\mathrm{t})} \qquad \frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C} \vdash \mathrm{u}' \rhd u'}{\mathcal{C} \vdash \mathrm{u} \to \mathrm{u}' \rhd u \to u'} \qquad \frac{}{\mathcal{C} \vdash \mathbf{int} \rhd \mathrm{int}}$$

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{t} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.t} \rhd \mathcal{S}(\mathrm{t})} \quad (1)$$

$$\boxed{\mathcal{C} \vdash \mathrm{B} \rhd \mathcal{L}} \qquad \frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C}, \mathrm{t} \rhd u \vdash \mathrm{B} \rhd \Lambda P.\mathcal{S} \quad \mathrm{t} \notin \mathcal{D}(\mathcal{S}) \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash \mathbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{B} \rhd \Lambda P.\mathrm{t} \rhd u, \mathcal{S}}$$

$$\frac{\alpha \notin \mathcal{V}(\mathcal{C}) \quad \mathcal{C}, \mathrm{t} \rhd \alpha \vdash \mathrm{B} \rhd \Lambda P.\mathcal{S} \quad \mathrm{t} \notin \mathcal{D}(\mathcal{S}) \quad \alpha \notin P}{\mathcal{C} \vdash \mathbf{type}\ \mathrm{t}; \mathrm{B} \rhd \Lambda\{\alpha\} \cup P.\mathrm{t} \rhd \alpha, \mathcal{S}} \quad (2)$$

$$\frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C}, \mathrm{x} : u \vdash \mathrm{B} \rhd \Lambda P.\mathcal{S} \quad \mathrm{x} \notin \mathcal{D}(\mathcal{S}) \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash \mathbf{val}\ \mathrm{x} : \mathrm{u}; \mathrm{B} \rhd \Lambda P.\mathrm{x} : u, \mathcal{S}}$$

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{S} \quad P \cap \mathcal{V}(\mathcal{C}) = \emptyset \quad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{B} \rhd \Lambda Q.\mathcal{S}' \quad \mathrm{X} \notin \mathcal{D}(\mathcal{S}') \quad Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset}{\mathcal{C} \vdash \mathbf{structure}\ \mathrm{X} : \mathrm{S}; \mathrm{B} \rhd \Lambda P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'}$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{B}} \rhd \Lambda\emptyset.\epsilon_{\mathcal{S}}}$$

$$\boxed{\mathcal{C} \vdash \mathrm{S} \rhd \mathcal{L}} \qquad \frac{\mathcal{C} \vdash \mathrm{B} \rhd \mathcal{L}}{\mathcal{C} \vdash \mathbf{sig}\ \mathrm{B}\ \mathbf{end} \rhd \mathcal{L}}$$

$$\boxed{\mathcal{C} \vdash \mathrm{e} : u} \quad \frac{\mathrm{x} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{x} : \mathcal{C}(\mathrm{x})} \qquad \frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C}, \mathrm{x} : u \vdash \mathrm{e} : u'}{\mathcal{C} \vdash \lambda \mathrm{x} : \mathrm{u.e} : u \to u'} \qquad \frac{\mathcal{C} \vdash \mathrm{e} : u' \to u \quad \mathcal{C} \vdash \mathrm{e}' : u'}{\mathcal{C} \vdash \mathrm{e}\ \mathrm{e}' : u}$$

$$\frac{}{\mathcal{C} \vdash \mathrm{i} : \mathrm{int}} \qquad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{x} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.x} : \mathcal{S}(\mathrm{x})}$$

$$\boxed{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}} \qquad \frac{\mathrm{X} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{X} : \mathcal{C}(\mathrm{X})} \quad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{X} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.X} : \mathcal{S}(\mathrm{X})}$$

Fig. 4. Common Denotation and Classification Judgements

4 Static Semantics

In this section we introduce two distinct static semantics for structure bodies and structure expressions. The systems rely on shared judgement forms relating Core types, signature bodies and signature expressions to their denotations, and Core expressions and structure paths to their types. The common judgements are shown in Figure 4. We can factor out these judgements because they do not generate any new *free* variables in their conclusions. Observe that the opaque type specifications in a signature expression give rise to the type parameters of the semantic signature it denotes (Rule 2).

4.1 Generative Semantics

Figure 5 presents a static semantics for structure bodies and expressions that employs *generative* classification judgements in the style of Standard ML [11].

Consider the form of the judgements $\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M$ and $\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow M$. The set of variables N is meant to capture a superset of the variables

$$\boxed{\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M} \qquad \frac{\mathcal{C} \vdash \mathrm{u} \triangleright u \quad \mathcal{C}, \mathrm{t} \triangleright u, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M}{\mathcal{C}, N \vdash \mathbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{b} : \mathrm{t} \triangleright u, \mathcal{S} \Rightarrow M}$$

$$\frac{\begin{array}{c}\alpha \notin N \qquad\qquad \mathcal{C}, \mathbf{t} \triangleright \alpha \vdash \mathrm{u} \triangleright u \\ \mathcal{C}, \mathrm{t} \triangleright \alpha, \mathrm{x} : u \to \alpha, \mathrm{x}' : \alpha \to u, N \cup \{\alpha\} \vdash \mathrm{b} : \mathcal{S}' \Rightarrow M\end{array}}{\mathcal{C}, N \vdash \mathbf{datatype}\ \mathrm{t}\ =\ u\ \mathbf{with}\ \mathrm{x}, \mathrm{x}'; \mathrm{b} : (\mathrm{t} \triangleright \alpha, \mathrm{x} : u \to \alpha, \mathrm{x}' : \alpha \to u, \mathcal{S}') \Rightarrow \{\alpha\} \cup M} \quad (3)$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : u \quad \mathcal{C}, \mathrm{x} : u, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M}{\mathcal{C}, N \vdash \mathbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} : \mathrm{x} : u, \mathcal{S} \Rightarrow M}$$

$$\frac{\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P \quad \mathcal{C}, \mathrm{X} : \mathcal{S}, N \cup P \vdash \mathrm{b} : \mathcal{S}' \Rightarrow Q}{\mathcal{C}, N \vdash \mathbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \mathrm{X} : \mathcal{S}, \mathcal{S}' \Rightarrow P \cup Q} \quad (4)$$

$$\frac{\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P \quad \mathcal{C}, \mathrm{X} : \mathcal{S}, N \cup P \vdash \mathrm{b} : \mathcal{S}' \Rightarrow Q}{\mathcal{C}, N \vdash \mathbf{local}\ \mathrm{X}\ =\ \mathrm{s}\ \mathbf{in}\ \mathrm{b} : \mathcal{S}' \Rightarrow P \cup Q}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \qquad P \cap N = \emptyset \qquad \mathcal{C}, \mathrm{X} : \mathcal{S}, N \cup P \vdash \mathrm{s} : \mathcal{S}' \Rightarrow Q \\ \mathcal{C}, \mathrm{F} : \forall P.\mathcal{S} \to \exists Q.\mathcal{S}', N \vdash \mathrm{b} : \mathcal{S}'' \Rightarrow M\end{array}}{\mathcal{C}, N \vdash \mathbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \mathbf{in}\ \mathrm{b} : \mathcal{S}'' \Rightarrow M}$$

$$\frac{}{\mathcal{C}, N \vdash \epsilon_{\mathrm{b}} : \epsilon_{\mathcal{S}} \Rightarrow \emptyset}$$

$$\boxed{\mathcal{C}, M \vdash \mathrm{s} : \mathcal{S} \Rightarrow N} \qquad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}}{\mathcal{C}, N \vdash \mathrm{sp} : \mathcal{S} \Rightarrow \emptyset} \qquad \frac{\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M}{\mathcal{C}, N \vdash \mathbf{struct}\ \mathrm{b}\ \mathbf{end} : \mathcal{S} \Rightarrow M}$$

$$\frac{\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P \quad \mathcal{C}(\mathrm{F}) > \mathcal{S}' \to \exists Q.\mathcal{S}'' \quad \mathcal{S} \succeq \mathcal{S}' \quad Q \cap (N \cup P) = \emptyset}{\mathcal{C}, N \vdash \mathrm{F(s)} : \mathcal{S}'' \Rightarrow P \cup Q} \quad (5)$$

$$\frac{\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda Q.\mathcal{S}' \quad \mathcal{S} \succeq \varphi(\mathcal{S}') \quad \mathcal{D}(\varphi) = Q}{\mathcal{C}, N \vdash \mathrm{s} : \mathrm{S} : \varphi(\mathcal{S}') \Rightarrow P}$$

$$\frac{\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda Q.\mathcal{S}' \quad \mathcal{S} \succeq \varphi(\mathcal{S}') \quad \mathcal{D}(\varphi) = Q \quad Q \cap N = \emptyset}{\mathcal{C}, N \vdash \mathrm{s} :> \mathrm{S} : \mathcal{S}' \Rightarrow Q} \quad (6)$$

Fig. 5. Generative Classification Judgements

generated so far. Classification produces, besides the semantic object $\mathcal{S}$, the set of variables M generated *during* the classification of the phrase b or s. The variable sets are threaded through classification trees in a global, state-like manner. This avoids any unsafe confusion of existing variables with the fresh variables generated by datatype definitions (Rule 3), functor applications (Rule 5), and opaque constraints (Rule 6). The generative nature of classification is expressed by the following property:

Property 1 (Generativity) *If* $\mathcal{C}, N \vdash \mathrm{b/s} \Rightarrow \mathcal{S}, M$ *then* $N \cap M = \emptyset$.[1]

Note that the sets of generated variables are *not* redundant. Suppose we deleted them from the classification judgements and replaced occurrences of N by $\mathcal{V}(\mathcal{C})$, so that variables are generated to be fresh with respect to just the current context instead of the state.

[1] When P is a predicate, we use the abbreviation $P(\mathrm{b/s})$ to mean $P(\mathrm{b})$ *and* $P(\mathrm{s})$.

For a counterexample, consider the following phrase:

$$
\begin{array}{l}
\textbf{structure X} = \textbf{struct datatype t} \ = \ \textbf{int with x}, \textbf{y end}; \\
\textbf{structure Y} = \textbf{struct structure X} \ = \ \textbf{struct} \ \ \textbf{end}; \\
\qquad\qquad\qquad\qquad\quad \textbf{datatype u} \ = \ \textbf{int} \rightarrow \textbf{int with x}', \textbf{y}' \\
\qquad\qquad\quad \textbf{end}; \\
\textbf{val z} = (\textbf{Y}.\textbf{y}' \ (\textbf{X}.\textbf{x} \ 1)) \ 2
\end{array}
$$

This phrase is unsound because the definition of **z** leads to the sad attempt of applying 1 to 2. The phrase should be rejected by a sound static semantics.

In the putatively simpler, state-less semantics, we only require that the type variables chosen for **t** and **u** are distinct from the variables free in the context of their respective definitions. The annotated phrase shows what can go wrong:

$$
\begin{array}{l}
\overset{\emptyset}{\lceil}\textbf{structure X} = \textbf{struct} \ \overset{\emptyset}{\lceil}\textbf{datatype t}_{\alpha} \ = \ \textbf{int with x}, \textbf{y end}; \\
\overset{\{\alpha\}}{\lceil}\textbf{structure Y} = \textbf{struct} \ \overset{\{\alpha\}}{\lceil}\textbf{structure X} \ = \ \textbf{struct} \ \ \textbf{end}; \\
\qquad\qquad\qquad\qquad\quad \overset{\emptyset}{\lceil}\textbf{datatype u}_{\alpha} \ = \ \textbf{int} \rightarrow \textbf{int with x}', \textbf{y}' \\
\qquad\qquad\quad \textbf{end}; \\
\overset{\{\alpha\}}{\lceil}\textbf{val z} = (\textbf{Y}.\textbf{y}'_{\alpha \rightarrow \textbf{int} \rightarrow \textbf{int}}(\textbf{X}.\textbf{x}_{\textbf{int} \rightarrow \alpha} \ 1)_{\alpha} \ )_{\textbf{int} \rightarrow \textbf{int}} \ 2
\end{array}
$$

Assuming an initially empty context, we have annotated the beginning of each structure body b with the set N of variables free in the local context, using the notation $\overset{N}{\lceil}$ b, the defining occurrences of **t** and **u** are decorated with their denotations, and key subphrases with their types. The problem is that **t** and **u** are assigned the same type variable α, even though they must be distinguished. The problem arises because α, already set aside for **t**, no longer occurs free in the local context at the definition of **u**: the free occurrence is eclipsed by the shadow of the second definition of **X**. Thus the semantics may again choose α to represent **u**, and incorrectly accept the definition of **z**.

The generative semantics that uses a state maintains soundness as follows:

$$
\begin{array}{l}
\overset{\emptyset}{\downarrow} \textbf{structure X} = \textbf{struct} \ \overset{\emptyset}{\downarrow} \textbf{datatype t}_{\alpha} \ = \ \textbf{int with x}, \textbf{y}\overset{\{\alpha\}}{\uparrow} \textbf{end}; \\
\overset{\{\alpha\}}{\downarrow} \textbf{structure Y} = \textbf{struct} \ \overset{\{\alpha\}}{\downarrow} \textbf{structure X} \ = \ \textbf{struct} \ \ \textbf{end}; \\
\qquad\qquad\qquad\qquad\quad \overset{\{\alpha\}}{\downarrow} \textbf{datatype u}_{\beta} \ = \ \textbf{int} \rightarrow \textbf{int with x}', \textbf{y}'\overset{\{\beta\}}{\uparrow} \\
\qquad\qquad\quad \textbf{end}; \\
\overset{\{\alpha,\beta\}}{\downarrow} \textbf{val z} = \underline{(\textbf{Y}.\textbf{y}'_{\beta \rightarrow \textbf{int} \rightarrow \textbf{int}}(\textbf{X}.\textbf{x}_{\textbf{int} \rightarrow \alpha} \ 1)_{\alpha} \ )} \ 2
\end{array}
$$

Assuming an initially empty context and state, we have indicated, at the beginning of each structure body b, the state N of variables generated so far, and, at its end, the variables M generated during its classification. We use the notation $\overset{N}{\downarrow}$ b $\overset{M}{\uparrow}$, corresponding to a classification $\ldots, N \vdash \mathrm{b} : \ldots \Rightarrow M$. Observe that generated variables are accumulated in the state as we traverse the phrase.

At the definition of $\mathbf{u}$, α is recorded in the state, even though it no longer occurs free in the current context, forcing the choice of a distinct variable β. In turn, this leads to the detection of the type violation, which is underlined.

These observations motivate:

Definition 4 (Rigidity) *$\mathcal{C}$ is* rigid *w.r.t. N, written $\mathcal{C}, N$* **rigid***, if and only if $\mathcal{V}(\mathcal{C}) \subseteq N$.*

As long as we start with $\mathcal{C}, N$ **rigid**, as a consequence of Property 1, those variables in M resulting from the classification of b and s will never be confused with variables visible in the context, even if these are temporarily hidden by bindings added to $\mathcal{C}$ during sub-classifications.

A similar example motivates the generativity of functor application. Consider this unsound phrase that applies 1 to 2 in the definition of $\mathbf{z}$:

functor F(**X**: **sig type t end**) = **struct datatype u** = **X**.**t with x**, **y end**
in
structure Y = **F**((**struct type t** = **int end**));
structure Z = **F**((**struct type t** = **int** → **int end**));
val z = (**Z**.**y** (**Y**.**x** 1)) 2

In a semantics with non-generative functors, we would simply add the variables returned by a functor's body to the state at the functor's definition, omitting the generation of fresh variables each time it is applied. Then each application of the functor would return the same generative types. In our example, this means that the types **Y**.**u** and **Z**.**u** would be identified, allowing the unsound definition of **z** to be accepted. In the generative semantics, each application of **F** returns new types, so that **Y**.**u** and **Z**.**u** are distinguished and the definition of **z** is correctly rejected. Observe that the definition of **u** depends on the functor argument's opaque type component **t**, whose realisation can vary with each application of **F**. The non-generative semantics for functors is unsound because it does not take account of this dependency; the generative semantics does.

4.2 Type-Theoretic Semantics

To a type theorist, the generative judgements appear odd. The intrusion of the state imposes a procedural ordering on the premises of the generative rules that is in contrast with the declarative, compositional formulation of typing rules in Type Theory. The fact that the type of the term may contain "new" free type variables, that do not occur free in the context, is peculiar (conventional type theories enjoy the free variable property: the type of a term is closed with respect to the variables occurring free in the context). Perhaps for this reason, generativity has developed its own mystique and its own terminology. In Standard ML [11], type variables are called "type names" to stress their persistent, generative nature. Generativity is often presented as an extra-logical device, useful for programming language type systems, but distinct from more traditional

$$\boxed{\mathcal{C} \vdash \mathrm{b} : \mathcal{X}} \qquad \frac{\mathcal{C} \vdash \mathrm{u} \triangleright u \quad \mathcal{C}, \mathrm{t} \triangleright u \vdash \mathrm{b} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash \mathbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{b} : \exists P.\mathrm{t} \triangleright u, \mathcal{S}}$$

$$\frac{\begin{array}{cc} \alpha \notin \mathcal{V}(\mathcal{C}) & \mathcal{C}, \mathbf{t} \triangleright \alpha \vdash \mathrm{u} \triangleright u \\ \mathcal{C}, \mathrm{t} \triangleright \alpha, \mathrm{x} : u \to \alpha, \mathrm{x}' : \alpha \to u \vdash \mathrm{b} : \exists P.\mathcal{S}' & P \cap (\{\alpha\} \cup \mathcal{V}(u)) = \emptyset \end{array}}{\mathcal{C} \vdash \mathbf{datatype}\ \mathrm{t}\ =\ u\ \mathbf{with}\ \mathrm{x}, \mathrm{x}'; \mathrm{b} : \exists \{\alpha\} \cup P.(\mathrm{t} \triangleright \alpha, \mathrm{x} : u \to \alpha, \mathrm{x}' : \alpha \to u, \mathcal{S}')} \quad (7)$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : u \quad \mathcal{C}, \mathrm{x} : u \vdash \mathrm{b} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash \mathbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} : \exists P.\mathrm{x} : u, \mathcal{S}}$$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(\mathcal{C}) = \emptyset \quad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{b} : \exists Q.\mathcal{S}' \quad Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset}{\mathcal{C} \vdash \mathbf{structure}\ \mathrm{X}\ =\ \mathrm{s};\mathrm{b} : \exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'} \quad (8)$$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(\mathcal{C}) = \emptyset \quad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{b} : \exists Q.\mathcal{S}' \quad Q \cap P = \emptyset}{\mathcal{C} \vdash \mathbf{local}\ \mathrm{X}\ =\ \mathrm{s}\ \mathbf{in}\ \mathrm{b} : \exists P \cup Q.\mathcal{S}'} \quad (9)$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \qquad P \cap \mathcal{V}(\mathcal{C}) = \emptyset \qquad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{s} : \mathcal{X} \\ \mathcal{C}, \mathrm{F} : \forall P.\mathcal{S} \to \mathcal{X} \vdash \mathrm{b} : \mathcal{X}' \end{array}}{\mathcal{C} \vdash \mathbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \mathbf{in}\ \mathrm{b} : \mathcal{X}'} \quad (10)$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \exists \emptyset.\epsilon_{\mathcal{S}}}$$

$$\boxed{\mathcal{C} \vdash \mathrm{s} : \mathcal{X}} \qquad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}}{\mathcal{C} \vdash \mathrm{sp} : \exists \emptyset.\mathcal{S}} \qquad \frac{\mathcal{C} \vdash \mathrm{b} : \mathcal{X}}{\mathcal{C} \vdash \mathbf{struct}\ \mathrm{b}\ \mathbf{end} : \mathcal{X}}$$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(\mathcal{C}(\mathrm{F})) = \emptyset \quad \mathcal{C}(\mathrm{F}) > \mathcal{S}' \to \exists Q.\mathcal{S}'' \quad \mathcal{S} \succeq \mathcal{S}' \quad Q \cap P = \emptyset}{\mathcal{C} \vdash \mathrm{F}(\mathrm{s}) : \exists P \cup Q.\mathcal{S}''}$$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda Q.\mathcal{S}' \quad P \cap \mathcal{V}(\Lambda Q.\mathcal{S}') = \emptyset \quad \mathcal{S} \succeq \varphi(\mathcal{S}') \quad \mathcal{D}(\varphi) = Q}{\mathcal{C} \vdash \mathrm{s} : \mathrm{S} : \exists P.\varphi(\mathcal{S}')}$$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda Q.\mathcal{S}' \quad P \cap \mathcal{V}(\Lambda Q.\mathcal{S}') = \emptyset \quad \mathcal{S} \succeq \varphi(\mathcal{S}') \quad \mathcal{D}(\varphi) = Q}{\mathcal{C} \vdash \mathrm{s} :> \mathrm{S} : \exists Q.\mathcal{S}'}$$

Fig. 6. Type-Theoretic Classification Judgements

type-theoretic constructs. In this section, we show how to replace the generative judgments by more declarative, type-theoretic ones.

Figure 6 presents an alternative static semantics for structure bodies and expressions, defined by the judgements $\mathcal{C} \vdash \mathrm{b} : \mathcal{X}$ and $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$. Rather than maintaining a global state of variables threaded through classifications, we classify structure bodies and expressions using *existential structures*.

The key idea is to replace *global* generativity with the introduction and elimination of existential types — in essence: *local* generativity. In the rules, the side conditions on bound variables prevent capture of free variables in the usual way. Because they are bound, the variables can always be renamed to satisfy the side conditions. For intuition, we explain some of the rules:

(**datatype** t $=$ u **with** $\mathrm{x}, \mathrm{x}'; \mathrm{b}$): The denotation of u is determined in the context extended with the recursive assumption that t denotes α, where α is a hypothetical type represented by a variable that is fresh for $\mathcal{C}$. This determines the types of the constructor x and the destructor x′ that are added to the context before classifying the body b. Provided b has existential structure $\exists P.\mathcal{S}$, which may contain occurrences of α, we conceptually eliminate the existential quantification over $\mathcal{S}$, introducing the hypothetical types P, extend the record of components t, x and x′ by $\mathcal{S}$ and then existentially quantify over both the hypothetical type α and the hypothetical types P we just introduced.

(**structure** X = s;b): Provided s has existential structure $\exists P.\mathcal{S}$, we eliminate the existential, introducing the hypothetical types P, and classify b in the context extended with the assumption X : $\mathcal{S}$ to obtain the existential structure $\exists Q.\mathcal{S}'$ of b. Now $\exists Q.\mathcal{S}'$ may contain some of the hypothetical types in P that should not escape their scope. We eliminate this existential, extend the component X : $\mathcal{S}$ by $\mathcal{S}'$ and existentially quantify over the hypothetical types $P \cup Q$.

(**functor** F (X : S) = s **in** b): The signature expression S denotes a family of semantic structures, $\Lambda P.\mathcal{S}$. For every φ with $\mathcal{D}(\varphi) = P$, F should be applicable at any enrichment, i.e. subtype, of $\varphi(\mathcal{S})$. To this end, we classify the body s of F in the context extended with the assumption X : $\mathcal{S}$. By requiring that P is a locally fresh choice of type variables, we know that $\mathcal{S}$ is a *generic* structure matching $\Lambda P.\mathcal{S}$, and that variables in P act as formal type parameters during the classification of the body. Classifying s yields an existential structure $\mathcal{X}$ that may contain occurrences of the parameters P. If this succeeds for a generic choice of parameters, it will also succeed for any realisation of these parameters[2]. We discharge the type parameters by universal quantification over P and add the assumption that F has the polymorphic type $\forall P.\mathcal{S} \to \mathcal{X}$ to the context. The scope b of the functor definition determines the type $\mathcal{X}'$ of the entire phrase.

(F(s)): Provided s has existential structure $\exists P.\mathcal{S}$, we locally eliminate the quantifier and choose an appropriate instance $\mathcal{S}' \to \exists Q.\mathcal{S}'$ of the functor's type. This step corresponds to eliminating the functor's polymorphism by choosing a realisation φ of its type parameters. The functor may applied if the actual argument's type $\mathcal{S}$ enriches the instance's domain $\mathcal{S}'$, i.e. provided $\mathcal{S}$ is a subtype of $\mathcal{S}'$. The range $\exists Q.\mathcal{S}''$ of the instance determines the type of the application, and may propagate some of the hypothetical types in P via the implicit realisation φ. To prevent these from escaping their scope, we abstract them by extending the existential quantification over $\mathcal{S}''$ to cover both P and Q.

(s : S): Provided s has existential type $\exists P.\mathcal{S}$ and S denotes, we first eliminate the existential quantification and then check that $\mathcal{S}$ matches the denotation of S. The denotation $\Lambda Q.\mathcal{S}'$ describes a family of semantic structures and the requirement is that the type $\mathcal{S}$ of the structure expression is a subtype of some member $\varphi(\mathcal{S}')$ of this family. Since φ is applied to $\mathcal{S}'$ in the conclusion $\exists P.\varphi(\mathcal{S}')$, the actual denotations of type components that have opaque specifications in S are preserved: however, the visibility of some components of s may be curtailed. The realised structure $\varphi(\mathcal{S}')$ may mention hypothetical types in P. Existentially quantifying over P prevents them from escaping their scope.

[2] (it can be shown that derivations are closed under realisation, hence for any φ with domain P, because $\mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{s} : \mathcal{X}$ we also know that $\varphi((\mathcal{C}, \mathrm{X} : \mathcal{S})) \vdash \mathrm{s} : \varphi(\mathcal{X})$ and this is equivalent to $\mathcal{C}, \mathrm{X} : \varphi(\mathcal{S}) \vdash \mathrm{s} : \varphi(\mathcal{X})$, since $P \cap \mathcal{V}(\mathcal{C}) = \emptyset$)

(s :> S): We proceed as in the previous case, but the type of s :> S is $\exists Q.\mathcal{S}'$, not $\exists P.\varphi(\mathcal{S}')$. Introducing the existential quantification over Q hides the realisation, rendering type components specified opaquely in S abstract.

Before we can state our main result we shall need one last concept:

Definition 5 (Ground Functors and Contexts) *A semantic functor* $\mathcal{F} \equiv \forall P.\mathcal{S} \rightarrow \mathcal{X}$ *is* ground*, written* $\vdash \mathcal{F}$ **Gnd** *if and only if* $P \subseteq \mathcal{V}(\mathcal{S})$*. A context* $\mathcal{C}$ *is ground, written* $\vdash \mathcal{C}$ **Gnd***, precisely when all the semantic functors in its range are ground.*

The ground property of a semantic functor $\mathcal{F}$ ensures that whenever we apply a functor of this type, the free variables of the range are either propagated from the actual argument, or were already free in $\mathcal{F}$. With this observation one can prove the following free variable lemma:

Lemma 1 (Free Vars) *If* $\vdash \mathcal{C}$ **Gnd** *then* $\mathcal{C} \vdash \mathrm{b/s} : \mathcal{X}$ *implies* $\mathcal{V}(\mathcal{X}) \subseteq \mathcal{V}(\mathcal{C})$*.*

Note that the ground property of contexts is preserved as an invariant of the classification rules. We only need to impose it when reasoning about classifications derived with respect to an arbitrary context, which might be non-ground.

We can revisit the example of Section 4.1 to demonstrate how our alternative semantics maintains soundness, without relying on a global state of generated type variables. Assume the initial context is empty. We indicate the existential types of the defining structure expressions using the notation $\overset{\mathcal{X}}{\lceil}\mathrm{s}$, and the types of the identifiers **X** and **Y** in the context using the notation $\underset{\mathcal{S}}{\lfloor}\mathrm{X}$. We also indicate the type variables chosen to represent **t** and **u** at their point of definition:

$$
\begin{array}{l}
\mathbf{structure}\ \underset{(\mathbf{t}\triangleright\alpha,\mathbf{x}:\mathbf{int}\rightarrow\alpha,\mathbf{y}:\alpha\rightarrow\mathbf{int})}{\lfloor\mathbf{X}} = \overset{\exists\{\alpha\}.(\mathbf{t}\triangleright\alpha,\mathbf{x}:\mathbf{int}\rightarrow\alpha,\mathbf{y}:\alpha\rightarrow\mathbf{int})}{\lceil\mathbf{struct}\ \mathbf{datatype}\ \mathbf{t}_\alpha\ =\ \mathbf{int}\ \mathbf{with}\ \mathbf{x},\mathbf{y}\ \mathbf{end}};\\
\mathbf{structure}\ \underset{(\mathbf{X}:\epsilon_{\mathcal{S}},\mathbf{u}\triangleright\beta,\mathbf{x}':(\mathbf{int}\rightarrow\mathbf{int})\rightarrow\beta,\mathbf{y}':\beta\rightarrow\mathbf{int}\rightarrow\mathbf{int})}{\lfloor\mathbf{Y}} = \overset{\exists\{\alpha\}.(\mathbf{X}:\epsilon_{\mathcal{S}},\mathbf{u}\triangleright\alpha,\mathbf{x}':(\mathbf{int}\rightarrow\mathbf{int})\rightarrow\alpha,\mathbf{y}':\alpha\rightarrow\mathbf{int}\rightarrow\mathbf{int})}{\lceil\mathbf{struct}\ \mathbf{structure}\ \mathbf{X}\ =\ \mathbf{struct}\ \ \mathbf{end}};\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{datatype}\ \mathbf{u}_\alpha\ =\ \mathbf{int}\rightarrow\mathbf{int}\ \mathbf{with}\ \mathbf{x}',\mathbf{y}'\\
\qquad\qquad\qquad\quad \mathbf{end};\\
\mathbf{val}\ \mathbf{z} = \underline{(\mathbf{Y}.\mathbf{y}'_{\beta\rightarrow\mathbf{int}\rightarrow\mathbf{int}}(\mathbf{X}.\mathbf{x}_{\mathbf{int}\rightarrow\alpha}\ 1)_\alpha\ )}\ 2
\end{array}
$$

The existential type of the structure expression defining **X** is:

$$\exists\{\alpha\}.(\mathbf{t} \triangleright \alpha, \mathbf{x} : \mathbf{int} \rightarrow \alpha, \mathbf{y} : \alpha \rightarrow \mathbf{int}).$$

Since α is fresh for the empty context, we can eliminate this existential quantifier directly so that, after the definition of **X**, the context of **Y** contains a free occurrence of α. As in the unsound state-less semantics discussed in Section 4.1, we are free to re-use α to represent **u** at the definition of **u**, because α no longer occurs in the context after the second definition of **X**. However, inspecting the existential type,

$$\mathcal{X} \equiv \exists\{\alpha\}.(\mathbf{X} : \epsilon_{\mathcal{S}}, \mathbf{u} \triangleright \alpha, \mathbf{x}' : (\mathbf{int} \rightarrow \mathbf{int}) \rightarrow \alpha, \mathbf{y}' : \alpha \rightarrow \mathbf{int} \rightarrow \mathbf{int}),$$

of the structure expression defining $\mathbf{Y}$, we can see that this variable is distinguished from the free occurrence of α in the context by the fact that it is existentially bound. Before we can extend the context with the type of $\mathbf{Y}$, we need to eliminate this existential quantifier. The first side-condition of Rule 8 requires that we avoid capturing the free occurrence of α in the context of $\mathbf{Y}$. To do this, it is necessary to choose a renaming of $\mathcal{X}$, in this case

$$\exists\{\beta\}.(\mathbf{X} : \epsilon_{\mathcal{S}}, \mathbf{u} \triangleright \beta, \mathbf{x}' : (\mathbf{int} \to \mathbf{int}) \to \beta, \mathbf{y}' : \beta \to \mathbf{int} \to \mathbf{int}),$$

for a variable β that is *locally* fresh for the context of $\mathbf{Y}$, and, in particular, distinct from α. After eliminating the renamed quantifier and extending the context with the type of $\mathbf{Y}$, the abstract types $\mathbf{X}.\mathbf{t}$ and $\mathbf{Y}.\mathbf{u}$ are correctly distinguished by α and β, catching the underlined type violation in the definition of $\mathbf{z}$.

5 Main Result

Having defined our systems, we can now state the main result of the paper:

Theorem 1 (Main Result) *Provided* $\vdash \mathcal{C}$ **Gnd** *and* $\mathcal{C}, N$ **rigid***:*

Completeness *If* $\mathcal{C}, N \vdash \mathrm{b/s} \Rightarrow \mathcal{S}, M$ *then* $\mathcal{C} \vdash \mathrm{b/s} : \exists M.\mathcal{S}$.
Soundness *If* $\mathcal{C} \vdash \mathrm{b/s} : \mathcal{X}$ *then there exist* $\mathcal{S}$ *and* M *such that* $\mathcal{C}, N \vdash \mathrm{b/s} \Rightarrow \mathcal{S}, M$ *with* $\mathcal{X} = \exists M.\mathcal{S}$.

An operational view of the systems in Figures 5 and 6 is that we have replaced the notion of *global* generativity by *local* generativity and the ability to rename bound variables when necessary. The proof of completeness is easy because a variable that is globally fresh will certainly be locally fresh, enabling a straightforward construction of a corresponding state-less derivation.

Proof (Completeness). By strong induction on the generative classification rules. We only describe the case for structure definitions, the others are similar:

$\boxed{\text{Rule 4}}$ Assume premises $\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P$ (i), $\mathcal{C}, \mathrm{X} : \mathcal{S}, N \cup P \vdash \mathrm{b} : \mathcal{S}' \Rightarrow Q$ (ii) and induction hypotheses $\vdash \mathcal{C}$ **Gnd** $\supset \mathcal{C}, N$ **rigid** $\supset \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$ (iii) and $\vdash \mathcal{C}, \mathrm{X} : \mathcal{S}$ **Gnd** $\supset \mathcal{C}, \mathrm{X} : \mathcal{S}, N \cup P$ **rigid** $\supset \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{b} : \exists Q.\mathcal{S}'$ (iv). Suppose $\vdash \mathcal{C}$ **Gnd** (v) and $\mathcal{C}, N$ **rigid** (vi). Now by induction hypothesis (iii) on (v) and (vi) we obtain $\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$ (vii). Property 1 of (i), together with (vi), ensures that $P \cap \mathcal{V}(\mathcal{C}) = \emptyset$ (viii). Clearly (v) extends to $\vdash \mathcal{C}, \mathrm{X} : \mathcal{S}$ **Gnd** (ix). Lemma 1 on (v) and (vii) ensures $\mathcal{V}(\exists P.\mathcal{S}) \subseteq \mathcal{V}(\mathcal{C})$. It follows from (vi) that $\mathcal{V}(\mathcal{S}) \subseteq N \cup P$ (x) and consequently $(\mathcal{C}, \mathrm{X} : \mathcal{S}), N \cup P$ **rigid** (xi). Applying induction hypothesis (iv) to (ix) and (xi) yields $\mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{b} : \exists Q.\mathcal{S}'$ (xii). Property 1 of (ii) ensures $Q \cap (N \cup P) = \emptyset$ which, together with (x), entails $Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset$ (xiii). Rule 8 on (vii), (viii), (xii) and (xiii) derives the desired result $\mathcal{C} \vdash$ **structure** $\mathrm{X} = \mathrm{s;b} : \exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'$.

In the complete proof, Property 1 and Lemma 1 conspire to ensure the side conditions on existentially bound variables and hence that implicit renamings of these variables are never required.

5.1 Soundness

Soundness is more difficult to prove, because the state-less rules in Figure 6 only requires sub derivations to hold for *particular* choices of locally fresh variables. A variable may be locally fresh without being globally fresh, foiling naive attempts to construct a generative derivation from a state-less derivation.

To address this problem, we introduce a modified formulation of the state-less classification judgements with the judgement forms $\mathcal{C} \vdash' \mathrm{b} : \mathcal{X}$ and $\mathcal{C} \vdash' \mathrm{s} : \mathcal{X}$ that have similar rules but with stronger premises. Instead of requiring premises to hold for *particular* choices of fresh variables, the modified rules require them to hold for *every* choice of variables. To express these rules, we define the concept of a *renaming* $\pi \in \mathit{Var} \stackrel{\text{fin}}{\to} \mathit{Type}$ that is similar to a realisation, but simply maps type variables to type variables. The operation of applying a renaming to a semantic object $\mathcal{O}$, written $\pi\langle\mathcal{O}\rangle$, is extended to all semantics objects in a way that avoids the capture of free variables by bound variables. For instance, the modified version of Rule 8 receives a stronger second premise:

$$\frac{\mathcal{C} \vdash' \mathrm{s} : \exists P.\mathcal{S} \quad \forall \pi.\mathcal{D}(\pi) = P \supset \mathcal{C}, \mathrm{X} : \pi\langle\mathcal{S}\rangle \vdash' \mathrm{b} : \pi\langle\exists Q.\mathcal{S}'\rangle \quad Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset}{\mathcal{C} \vdash' \textbf{structure}\ \mathrm{X} = \mathrm{s;b} : \exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'}$$

Similar changes are required to Rules 7, 9 and 10 that extend the context with objects containing locally fresh variables. The generalised premises make it easy to construct a generative derivation from the derivation of a generalised judgement. Although these rules are not finitely branching, the judgements are well-founded and amenable to inductive arguments. This technique is adapted from [10].

Our proof strategy is to first show that any derivation of a state-less judgement gives rise to a corresponding derivation of a generalised judgement:

Lemma 2 *If* $\vdash \mathcal{C}$ **Gnd** *and* $\mathcal{C} \vdash \mathrm{b/s} : \mathcal{X}$ *then* $\mathcal{C} \vdash' \mathrm{b/s} : \mathcal{X}$.

We then show that any derivation of a generalised judgement gives rise to a corresponding generative derivation:

Lemma 3 *If* $\vdash \mathcal{C}$ **Gnd** *and* $\mathcal{C} \vdash' \mathrm{b/s} : \mathcal{X}$ *then, for any* N *satisfying* $\mathcal{C}, N$ **rigid**, *there exist* $\mathcal{S}$ *and* M *such that* $\mathcal{C}, N \vdash \mathrm{b/s} \Rightarrow \mathcal{S}, M$, *with* $\mathcal{X} = \exists M.\mathcal{S}$.

The proofs require stronger induction hypotheses and are technically involved. Further details can be found in the author's thesis [12].

6 Contribution

Theorem 1 is an equivalence result, but we propose that the state-less semantics provides a better *conceptual* understanding of the type structure of Standard ML.

The core type phrase sp.t, which introduces a dependency of Mini-SML's type syntax on its term syntax, suggests that Mini-SML's type structure is based on first-order dependent types. However, arguing from our semantics, we can show that first-order dependent types play no role in the semantics.

Compare the syntactic types of Mini-SML with their semantics counterparts, the semantic objects that are used to classify Mini-SML terms. Where type phrases allow occurrences of type identifiers and term dependent projections sp.t, semantic types instead allow occurrences of *type variables* $\alpha \in \mathit{Var}$. Type variables range over semantic types and are thus *second-order* variables. While the component specifications of a signature expression are dependent, in that subsequent specifications can refer to term identifiers specified previously in the body, the body of a semantic signature is just an unordered finite map, with no dependency between its components: the identifiers in a semantic structure, like the field names of record types, do not have scope. Thus there is a clear distinction between syntactic types and semantics objects: where syntactic types have first-order dependencies on term identifiers, semantic types have second-order dependencies on type variables.

The reduction of first-order to second-order dependencies is achieved by Mini-SML's denotation judgements. In particular, the denotation of the term dependent type sp.t is determined by the type, not the value, of the term sp. In conjunction with Rule 2, that assigns type variables to opaque type specifications, Rule 1 reduces the first-order dependencies of syntactic types on terms to second-order dependencies of semantic types on type variables.

We can illustrate this reduction by comparing the following signature expression with its denotation:

$$
\begin{array}{ll}
\begin{array}{l}
\textbf{sig structure X} : \textbf{sig type t} \\
\qquad\qquad\qquad\quad \textbf{end}; \\
\quad \textbf{structure Y} : \textbf{sig type u}; \\
\qquad\qquad\qquad\qquad \textbf{type v} = \textbf{X}.\textbf{t} \to \textbf{u} \\
\qquad\qquad\qquad\quad \textbf{end}; \\
\quad \textbf{val y}: \textbf{X}.\textbf{t} \to \textbf{Y}.\textbf{v} \\
\textbf{end}
\end{array}
&
\begin{array}{l}
\Lambda\{\alpha,\beta\}.\,(\mathbf{X} : (\mathbf{t} \triangleright \alpha), \\
\qquad\quad \mathbf{Y} : (\mathbf{u} \triangleright \beta, \\
\qquad\qquad\quad \mathbf{v} \triangleright \alpha \to \beta), \\
\qquad\quad \mathbf{y} : \alpha \to \alpha \to \beta)
\end{array}
\end{array}
$$

The opaque types **t** and **u** are represented by type variables α and β; the dependency on the terms **X** and **Y** in the specifications of **v** and **y** have disappeared.

As another example, let S be the above signature expression and consider the following functor and its type:

$$
\begin{array}{ll}
\begin{array}{l}
\textbf{functor F}(\mathbf{Z} : \mathrm{S}) = \\
\qquad \textbf{struct type w} = \mathbf{Z}.\mathbf{X}.\mathbf{t} \to \mathbf{Z}.\mathbf{Y}.\mathbf{v}; \\
\qquad\qquad \textbf{val z} = \mathbf{Z}.\mathbf{y} \\
\qquad \textbf{end}
\end{array}
&
\begin{array}{l}
\forall\{\alpha,\beta\}.(\mathbf{X} : (\mathbf{t} \triangleright \alpha), \\
\qquad\quad \mathbf{Y} : (\mathbf{u} \triangleright \beta, \\
\qquad\qquad\quad \mathbf{v} \triangleright \alpha \to \beta), \\
\qquad\quad \mathbf{y} : \alpha \to \alpha \to \beta) \\
\qquad \to \exists\emptyset.(\mathbf{w} \triangleright \alpha \to \alpha \to \beta, \\
\qquad\qquad \mathbf{z} : \alpha \to \alpha \to \beta)
\end{array}
\end{array}
$$

F returns the type **w** whose definition depends on the term argument **Z**. In the semantic object, this first-order dependency has been eliminated, in favour of a second-order dependency on the functor's type parameters α and β.

Our choice of binding notation and the reformulation of the generative classification rules further underline the fact that Mini-SML, and thus Standard ML, is based on a purely second-order type theory. In this interpretation, signatures are types parameterised on type variables, functor are polymorphic functions whose types have universally quantified type variables, and structure expressions have types with existentially quantified type variables. A universal quantifier is explicitly introduced when a functor is defined and silently eliminated when it is applied. An existential quantifier is explicitly introduced by a datatype definition or an opaque signature constraint, and silently eliminated and re-introduced at other points in the semantics. Allowing a functor's actual argument and a constraint's structure expression to have a richer type is an appeal to subtyping that can easily be factored into a separate subsumption rule as in traditional formalisations of subtyping in Type Theory. We have not done this to keep the classification rules syntax directed: this avoids admitting non-principal classifications and simplifies the statement and proof of soundness.

The style of semantics presented here scales naturally to both higher-order and first-class modules [12]. The higher-order extension is competive with, though subtly different from, the calculi of [1, 5, 6, 8]. Where these caculi have the advantage of syntactic types, ours has the advantage of enjoying principal types. The extension to first-class modules, which requires just three new Core constructs to specify, introduce and eliminate first-class module types, has a decidable type checking problem, unlike [1, 8]. Both extensions are compatible with ML-style type inference for the Core.

Acknowledgements: Many thanks to Don Sannella and Healfdene Goguen.

References

1. R. Harper, M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM Symp. Principles of Prog. Lang.*, 1994.
2. R. Harper, J. C. Mitchell. On the type structure of Standard ML. In *ACM Trans. Prog. Lang. Syst.*, volume 15(2), pages 211–252, 1993.
3. R. Harper, J. C. Mitchell, E. Moggi. Higher-order modules and the phase distinction. T. R. ECS-LFCS-90-112, Department of Computer Science, University of Edinburgh, 1990.
4. R. Harper, C. Stone. An Interpretation of Standard ML in Type Theory. T. R. CMU-CS-97-147, School of Computer Science,Carnegie Mellon University, 1997.
5. X. Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st Symp. Principles of Prog. Lang.*, pages 109–122. ACM Press, 1994.
6. X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd Symp. Principles of Prog. Lang.*, pages 142–153. ACM Press, 1995.
7. X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):1—32, 1996.
8. M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
9. D. MacQueen. Using dependent types to express modular structure. In *13th ACM Symp. on Principles of Prog. Lang.*, 1986.
10. J. McKinna, R. Pollack. Pure Type Sytems formalized. In *Proc. Int'l Conf. on Typed Lambda Calculi and Applications, Utrecht*, pages 289–305, 1993.
11. R. Milner, M. Tofte, R. Harper, D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
12. C. V. Russo. Types For Modules. PhD Thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1998.