J. Functional Programming **6** (5): 1–32, September 1996

A syntactic theory of type generativity and sharing

XAVIER LEROY
INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay, France

Abstract

This paper presents a purely syntactic account of type generativity and sharing — two key mechanisms in the SML module system — and shows its equivalence with the traditional stamp-based description of these mechanisms. This syntactic description recasts the SML module system in a more abstract, type-theoretic framework.

Capsule Review

The issue of "name equivalence" versus "structural equivalence" for statically typed languages has been a vexing problem in programming language design for decades. With the introduction of sophisticated modularity mechanisms such as are found in Standard ML, a straightforward account of type equivalence has remained elusive. In particular, the important notions of "type generativity", whereby module instances generate new types, and "sharing constraint", with which two types may be constrained to arise from the same instance, have proved difficult to formalize precisely. The formal definition of Standard ML provides an operational interpretation of typing, relying on the notion of a "stamp" to handle generativity and sharing. In this paper Leroy provides a declarative formulation of a type system that provides a natural and readily comprehensible account of these notions, and establishes the equivalence of the two approaches for the case of first-order module systems.

1 Introduction

First introduced to justify a posteriori the use of name equivalence in implementations of type-checkers, the notion of type generativity (the fact that, in some languages, data type definitions generate "new" types incompatible with any other types including data types with similar structure) has since emerged as a key mechanism to implement type abstraction — the important programming technique where a named type `t` is equipped with operations `f`, `g`, ... then the concrete implementation of `t` is hidden, leaving an abstract type `t` that can only be accessed through the operations `f`, `g`, ... (Liskov and Guttag, 1986).

Type generativity plays a crucial role in type abstraction, since making a type `t` abstract amounts to generating a new type `t` incompatible with any other type, including its concrete representation and other data types with similar structure.

Generativity ensures that only the operations provided over `t` can access its concrete representation.

Type generativity is therefore an essential feature of type abstraction; unfortunately, it is also one of the most mysterious and most difficult to define formally in a type system. In simple cases such as Modula-2's modules (Wirth, 1983), name equivalence provides a satisfactory notion of generativity: generated types are represented in type expressions by their names; uniqueness of names is ensured by suitable syntactic restrictions or by renamings. Unfortunately, this simple approach does not extend easily to more powerful module and type abstraction systems, in particular those that feature parameterized abstractions, known as *functors* in SML (MacQueen, 1984; Harper et al., 1987). The main reason is that if the result of a functor contains a generative type declaration, then a new type must be generated for each application of the functor. Otherwise, two different structures obtained by applying a functor to two different arguments would have compatible type components and therefore could access each other's representations, which violates abstraction.

Parameterized abstractions such as SML's functors raise other interesting issues in connection with type generativity. First, not all type components in functor results are generative: in many practical situations, they are simply inherited from some types in the functor argument, and the compatibility between the argument type and the result type must be preserved. Second, we should also account for SML's *sharing constraints*: the mechanism by which a functor with several arguments can require that some type components of its arguments are actually the same type, or in other terms that they have been generated at the same time (MacQueen, 1984; Harper et al., 1987). Therefore, a theory of type generativity must not go too far and e.g. systematically generate new types for all functor applications. Instead, it should maintain a suitable notion of type identity, where new types are generated when the programmer requires it, but the identities of existing types are correctly propagated otherwise.

Two approaches to type generativity and sharing have been investigated so far. The first approach, exemplified by the SML definition (Milner et al., 1990), formalizes the use of stamps to represent generative types (a common implementation practice) and extends it to the case of functors. The result is a calculus over stamps that captures the expected behavior of type generativity and can also express the related but more involving notions of structure generativity and sharing.

The second approach to type abstraction relies on name equivalence to account for type generativity: generative types are represented in the type algebra by free or existentially quantified type variables, which are structurally different from any other type expression except themselves; suitable restrictions over variable names ensure that two types generated at different times will always be represented by different variables. A number of type systems have been developed along these lines (Mitchell and Plotkin, 1988; Cardelli and Wegner, 1985; Cardelli, 1989; Cardelli and Leroy, 1990), more type-theoretic in flavor than the stamp-based descriptions. These type systems are relatively easy to extend and reason about, but generally fail to account for the expected behavior of type generativity: new types are generated in situations where the identity of an existing type should be preserved (MacQueen, 1986); moreover, sharing constraints are not accounted for.

The purpose of the present paper is to show that a simple extension of the type-theoretic approach — the introduction of type equalities in signatures, as proposed in (Harper and Lillibridge, 1994; Leroy, 1994) — succeeds in capturing a reasonable notion of type identity: we will prove that a type system for a simplified module language, derived from (Leroy, 1994), expresses exactly the same notion of type generativity and sharing as the SML stamp-based static semantics. Assuming that the latter captures "the" right notion of type generativity, as suggested by the large amount of practical experience gained with the SML module system, the equivalence result in this paper therefore proves that we have finally obtained a satisfactory type-theoretic description of type generativity and sharing.

The remainder of this paper is organized as follows. Section 2 introduces a skeletal module language and gives it a stamp-based static semantics in the style of the SML definition. Section 3 reviews the type systems for type abstraction and generativity that have been proposed so far. As an application of these ideas, section 4 gives a type system (without stamps) for the skeletal module language, subject to some syntactic restrictions. Section 5 defines a normalization process that circumvents the syntactic restrictions in a way compatible with the stamp-based static semantics. Section 6 finally proves the equivalence of the type system and the static semantics on normalized programs.

2 A stamp-based static semantics

The simplified module language TypModL used in this paper is derived from the ModL calculus introduced by Tofte *et al.* to study the related notions of structure sharing and generativity (Harper et al., 1987; Tofte, 1988; Tofte, 1992; Aponte, 1993). TypModL features generative and non-generative type declarations, structures, and first-order functors. The main simplification with respect to the SML module system is that structures have no value components, since the presence of values in structures is irrelevant to our study of type generativity. Also, sharing constraints between structures are not supported.

In the following grammar, t ranges over type identifiers, x over structure identifiers, and f over functor identifiers.

Programs:

$m ::= \varepsilon$		the empty program
	$\mid$ `structure` $x = s;\ m$	structure binding
	$\mid$ `functor` $f(x : S) = s;\ m$	functor binding

Structure expressions:

$s ::= p_s$		access to a structure
	$\mid$ `struct` d `end`	structure construction
	$\mid f(s)$	functor application

Structure body:

$d ::= \varepsilon \mid c; d$

Definitions:

c ::=	`type` $t = T$	type binding (non generative)
	$\mid$ `datatype` t	type creation (generative)
	$\mid$ `structure` $x = s$	structure binding
	$\mid$ `open` s	structure inclusion

Signature expressions:

S ::= `sig` D `end`

Signature body:

$D ::= \varepsilon \mid C; D$

Specifications:

C ::=	`type` t	type specification
	$\mid$ `structure` $x : S$	structure specification
	$\mid$ `sharing` $p_t = p'_t$	sharing constraint

Type expressions:

$T ::= p_t \mid T_1 \rightarrow T_2$

Structure paths:

$p_s ::= x \mid p_s.x$

Type paths:

$p_t ::= t \mid p_s.t$

A program is a sequence of structure and functor definitions. A structure contains definitions for types and sub-structures. Two kinds of type definitions are provided: non-generative, `type` $t = T$, which defines t as a synonym for the type expression T, and generative, `datatype` t, which creates a "new" type t. The latter is intended to model the `datatype` and `abstype` constructs in SML, which define new types with associated constructors or functions. Since we do not have values in this calculus, the constructors and functions associated with `datatype` and `abstype` are omitted.

The static semantics (compile-time checks) for this calculus is a direct adaptation of the static semantics given in the SML definition (Milner et al., 1990, chapter 5). It uses the "semantic" objects defined below to represent types and structures at compile-time. Stamps, written n, range over a countable set of identifiers†.

Types:	$\tau ::= n \mid \tau_1 \rightarrow \tau_2$	
Signatures:	$\Sigma ::= \{t \mapsto \tau;\ x \mapsto \Sigma\}$	
Functor signatures:	$\Phi ::= \forall N_1.\,(\Sigma_1, \forall N_2.\,\Sigma_2)$	N_1, N_2 are sets of stamps
Environments:	$\Gamma ::= \{t \mapsto \tau;\ x \mapsto \Sigma;\ f \mapsto \Phi\}$	

Types τ are either function types or stamps representing generated types. Signatures Σ are finite maps from type identifiers to types, and from structure identifiers to signatures. Environments Γ are similar, but also map functor identifiers to functor

† Stamps are called "names" in the SML definition (Milner et al., 1990). Following (MacQueen and Tofte, 1994), we prefer to call these semantic entities "stamps", and reserve "names" for syntactic entities (names of structure fields) that will be introduced in section 3.

signatures. Functor signatures $\forall N_1.(\Sigma_1, \forall N_2.\Sigma_2)$ are composed of two signatures and two sets of universally quantified names: the signature Σ_1 describes the expected shape for the argument, the signature Σ_2 describes the result structure, N_2 is the set of stamps that must be generated afresh at each application, and N_1 is the set of "flexible" stamps in the functor argument (those that can be instantiated to match the structure provided as argument). Functor signatures are identified modulo renaming of bound stamps.

The static semantics is defined by the inference rules in figure 1. The rules define several "elaboration judgements" of the form $\Gamma \vdash$ *syntactic object* $\Rightarrow$ *semantic object*, which check the well-formedness of syntactic objects in the elaboration environment Γ and return their representations as semantic objects. Some rules have an additional parameter W (a set of stamps), recording which stamps are already in use and guaranteeing that fresh stamps are generated for `datatype` declarations. We write $FS(\Sigma)$ for the set of stamps occurring free in the signature Σ.

The rules for elaborating type and structure expressions (rules 1–10) translate type and structure paths according to the environment Γ and build the corresponding semantic objects. The rule for `datatype` declarations (rule 8) assigns a fresh stamp n to the type being declared: n is chosen outside of W, the set of stamps already in use, then added to W for the elaboration of the remainder of the definition, which ensures that n will not be assigned later to another `datatype` declaration.

The most interesting rule is the rule for functor application (rule 5). Matching the structure argument against the functor argument signature involves two steps: an instantiation step (written $\leq$) where the flexible stamps in the argument and result signatures are replaced by types matching those in the argument structure, and an enrichment step (written $\succ$), which checks that the argument contains all the required components and possibly more. These two steps are formally defined as follows:

Definition 1 (Instantiation relation)
$(\Sigma'_1, \forall N'_2.\Sigma'_2) \leq \forall N_1.(\Sigma_1, \forall N_2.\Sigma_2)$ holds if there exists a substitution φ of types for stamps such that $\mathrm{Dom}(\varphi) \subseteq N_1$ and $\Sigma'_1 = \varphi(\Sigma_1)$ and $\forall N_2.\Sigma'_2 = \varphi(\forall N'_2.\Sigma_2)$.

Definition 2 (Enrichment relation)
$\Sigma_1 \succ \Sigma_2$ holds if $\mathrm{Dom}(\Sigma_1) \supseteq \mathrm{Dom}(\Sigma_2)$, and $\Sigma_1(t) = \Sigma_2(t)$ for all $t \in \mathrm{Dom}(\Sigma_2)$, and $\Sigma_1(x) \succ \Sigma_2(x)$ for all $x \in \mathrm{Dom}(\Sigma_2)$.

The constraint $N \cap (W \cup FS(\Sigma)) = \emptyset$ in rule 5 ensures that fresh stamps are assigned to the generative types in the functor result (Milner et al., 1990; Gran, 1995).

Elaboration of signature expressions (rules 11–15) is straightforward, except that the rule for `type` components (rule 13) allows any type, not only fresh stamps, to be assigned to the type identifier. This way, subsequent sharing constraints can be satisfied (rule 14) by suitable choice of these types.

As a consequence of rule 13, a signature expression can elaborate to many different signatures. However, some of these signatures are principal in the following sense: Σ is principal for S in Γ if $\Gamma \vdash S \Rightarrow \Sigma$ and, for all Σ' such that

Elaboration of paths: $\Gamma(p_s.x) = (\Gamma(p_s))(x) \qquad \Gamma(p_s.t) = (\Gamma(p_s))(t)$

Elaboration of type expressions:

$$\Gamma \vdash p_t \Rightarrow \Gamma(p_t) \quad (1) \qquad\qquad \frac{\Gamma \vdash T_1 \Rightarrow \tau_1 \qquad \Gamma \vdash T_2 \Rightarrow \tau_2}{\Gamma \vdash (T_1 \rightarrow T_2) \Rightarrow (\tau_1 \rightarrow \tau_2)} \quad (2)$$

Elaboration of structure expressions:

$$\Gamma, W \vdash p_s \Rightarrow \Gamma(p_s) \quad (3) \qquad\qquad \frac{\Gamma, W \vdash d \Rightarrow \Sigma}{\Gamma, W \vdash \texttt{struct}\ d\ \texttt{end} \Rightarrow \Sigma} \quad (4)$$

$$\frac{\Gamma, W \vdash s \Rightarrow \Sigma \qquad (\Sigma_1, \forall N.\, \Sigma_2) \leq \Gamma(f) \qquad \Sigma \succ \Sigma_1 \qquad N \cap (W \cup FS(\Sigma)) = \emptyset}{\Gamma, W \vdash f(s) \Rightarrow \Sigma_2} \quad (5)$$

$$\Gamma, W \vdash \varepsilon \Rightarrow \{\} \quad (6) \qquad\qquad \frac{\Gamma, W \vdash T \Rightarrow \tau \qquad \Gamma + \{t \mapsto \tau\},\ W \vdash d \Rightarrow \Sigma}{\Gamma, W \vdash (\texttt{type}\ t = T;\ d) \Rightarrow \{t \mapsto \tau\} + \Sigma} \quad (7)$$

$$\frac{n \notin W \qquad \Gamma + \{t \mapsto n\},\ W \cup \{n\} \vdash d \Rightarrow \Sigma}{\Gamma, W \vdash (\texttt{datatype}\ t;\ d) \Rightarrow \{t \mapsto n\} + \Sigma} \quad (8)$$

$$\frac{\Gamma, W \vdash s \Rightarrow \Sigma_1 \qquad \Gamma + \{x \mapsto \Sigma_1\},\ W \cup FS(\Sigma_1) \vdash d \Rightarrow \Sigma_2}{\Gamma, W \vdash (\texttt{structure}\ x = s;\ d) \Rightarrow \{x \mapsto \Sigma_1\} + \Sigma_2} \quad (9)$$

$$\frac{\Gamma, W \vdash s \Rightarrow \Sigma_1 \qquad \Gamma + \Sigma_1,\ W \cup FS(\Sigma_1) \vdash d \Rightarrow \Sigma_2}{\Gamma, W \vdash (\texttt{open}\ s;\ d) \Rightarrow \Sigma_1 + \Sigma_2} \quad (10)$$

Elaboration of signature expressions:

$$\frac{\Gamma \vdash D \Rightarrow \Sigma}{\Gamma \vdash (\texttt{sig}\ D\ \texttt{end}) \Rightarrow \Sigma} \quad (11) \qquad\qquad \Gamma \vdash \varepsilon \Rightarrow \{\} \quad (12)$$

$$\frac{\Gamma + \{t \mapsto \tau\} \vdash D \Rightarrow \Sigma}{\Gamma \vdash (\texttt{type}\ t;\ D) \Rightarrow \{t \mapsto \tau\} + \Sigma} \quad (13) \qquad \frac{\Gamma(p_t) = \Gamma(p'_t) \qquad \Gamma \vdash D \Rightarrow \Sigma}{\Gamma \vdash (\texttt{sharing}\ p_t = p'_t;\ D) \Rightarrow \Sigma} \quad (14)$$

$$\frac{\Gamma \vdash S \Rightarrow \Sigma_1 \qquad \Gamma + \{x \mapsto \Sigma_1\} \vdash D \Rightarrow \Sigma_2}{\Gamma \vdash (\texttt{structure}\ x : S;\ D) \Rightarrow \{x \mapsto \Sigma_1\} + \Sigma_2} \quad (15)$$

Elaboration of programs:

$$\Gamma, W \vdash \varepsilon \Rightarrow \texttt{ok} \quad (16) \qquad \frac{\Gamma, W \vdash s \Rightarrow \Sigma \qquad \Gamma + \{x \mapsto \Sigma\},\ W \cup FS(\Sigma) \vdash m \Rightarrow \texttt{ok}}{\Gamma \vdash (\texttt{structure}\ x = s;\ m) \Rightarrow \texttt{ok}} \quad (17)$$

$$\frac{\begin{array}{c}\Gamma \vdash S \Rightarrow \Sigma_1 \qquad \Sigma_1 \text{ is principal for } S \text{ in } \Gamma \qquad N_1 = FS(\Sigma_1) \setminus W \\ \Gamma + \{x \mapsto \Sigma_1\},\ W \cup FS(\Sigma_1) \vdash s \Rightarrow \Sigma_2 \qquad N_2 = FS(\Sigma_2) \setminus FS(\Sigma_1) \setminus W \\ \Gamma + \{f \mapsto \forall N_1.\, (\Sigma_1, \forall N_2.\, \Sigma_2)\},\ W \vdash m \Rightarrow \texttt{ok}\end{array}}{\Gamma, W \vdash (\texttt{functor}\ f(x : S) = s;\ m) \Rightarrow \texttt{ok}} \quad (18)$$

Fig. 1. Static semantics with stamps

$\Gamma \vdash S \Rightarrow \Sigma'$, there exists a substitution φ of types for stamps such that $\Sigma' = \varphi(\Sigma)$ and $\mathrm{Dom}(\varphi) \cap FS(\Gamma) = \emptyset$. Intuitively, a signature is principal for S if it captures exactly the sharing required by S, but no more. It is easy to prove that any signature expression that elaborates in Γ admits a principal signature in Γ (Tofte, 1988).

The rule for functor declarations (rule 18) represents the functor argument by its principal signature during the elaboration of the structure body. The stamps that are free in the principal signature but not used in Γ become the flexible stamps N_1 in the functor signature; the stamps N_2 that are free in the functor body signature but not in Γ nor in N_1 are the stamps N_2 that are generated at each application.

3 Type systems for type abstraction

In this section, we review some previously proposed type systems for type abstraction and progressively introduce the main ingredients of our type system. Unlike the static semantics, these systems are purely syntactic in nature: the typing rules involve only structure and signature expressions; no elaboration into richer semantic objects is required.

3.1 Existential types

Mitchell and Plotkin (1988) derived the first such type system from the observation that type abstraction has strong connections with second-order existential quantification in logic: a signature `sig type t; ... end` is viewed as the existential statement "there exists a type `t` such that ..."; a structure `struct type t=`τ`;` `... end`, as a constructive proof of this statement.

To access structure components, the Mitchell-Plotkin approach does not use projections ($s.t$ to refer to the component t of s) as in the SML modules, but instead a binding construct `open` s `as` D `in` e, modeled after $\exists$-elimination in constructive logic. This construct binds the components of structure s to the identifiers specified in D (a signature body), then evaluates the expression e in the enriched context. For the purpose of type-checking e, the type identifiers $t_1 \dots t_n$ declared in D are treated as free type variables in the type algebra, hence t_i is incompatible with any type except itself. This ensures that e is parametric in $t_1 \dots t_n$ and can safely be executed with any concrete implementation of $t_1 \dots t_n$.

Type abstraction is ensured by two crucial syntactic restrictions on the `open` construct. First, the type identifiers $t_1 \dots t_n$ bound by `open` must not be already bound in the current environment, to ensure the uniqueness of the types $t_1 \dots t_n$. Second, the identifiers $t_1 \dots t_n$ must not appear free in the type of the body e of the `open` construct.

This approach provides a simple and elegant treatment of type abstraction. Its main weakness is that it does not correctly preserve the identity of abstract types (Cardelli and MacQueen, 1988): if `open` is applied twice to the same structure, the two sets of type identifiers thus introduced will not match; hence, types are generated when structures are opened, not when they are created. As argued by

MacQueen (1986), this lack of a unique "witness" for each abstract type makes this approach inappropriate for modular programming.

3.2 The dot notation

In an attempt to address this issue, Cardelli (1988; 1989) has proposed a variant of the Mitchell-Plotkin approach where the `open` elimination construct is replaced by the "dot notation", that is, a projection-like elimination construct similar to SML's long identifiers and Modula-2's qualified identifiers: in a type context, $s.t$ refers to the type component t of structure s. Two type expressions $s.t$ and $s'.t$ are compatible if and only if s and s' are syntactically identical.

As in Mitchell and Plotkin's approach, type abstraction is here ensured via suitable syntactic restrictions. First, s in $s.t$ cannot be any structure expression, but is required to be a structure identifier x or structure path $x.x_1.x_2 \ldots x_n$, to ensure that its evaluation cannot generate new types. The following example illustrates what goes wrong if this restriction is lifted:

```
functor F(X: sig end) = struct datatype t; ... end;
... F(struct end).t ... F(struct end).t ...
```

The two applications of `F` should generate distinct types `t`, but the two occurrences of `F(struct end).t`, which are syntactically identical, would be considered as compatible types according to the equivalence rule for type expressions.

The other restrictions are similar to those for the `open` construct: functor parameters must not appear in the result signature of the functor (no dependent functor types); structure bindings (in structures or as functor parameters) must not rebind an already bound structure name. As an example of incorrect rebinding, assume the current environment is

```
structure s : (sig type t; val v:t end);
val x : s.t
```

and consider the definition

```
structure s = struct datatype t end
```

After typing the definition and entering `structure s : sig type t end` in the environment, `x` would appear to have type `s.t` (the newly generated type), which is semantically false. Therefore, rebindings must be avoided by prior renaming of bound identifiers, as with the `open` construct.

Projections and renamings

This necessary renaming of identifiers is problematic in conjunction with the dot notation. Identifiers bound inside a `struct ... end` should not be renamed, since the dot notation relies on their names to extract a component of the structure. For instance, if we rename `t` to `t'` in the structure

```
structure s = struct type t = int; ... end
```

then further references to `s.t` become invalid. (There is no sensible way to transform them into `s.t'`, since they are outside the scope of the binding `type t = int`.) One solution to this problem is to use positions instead of names to extract structure components (Cardelli and Leroy, 1990), but this makes signature subsumption problematic. A more general solution is to distinguish between names and identifiers (Leroy, 1994; Harper and Lillibridge, 1994). Each identifier has a name, but distinct identifiers may have the same name. Access in structures is by component name; type equivalence and references to bound variables are by identifier. We write t, x for type and structure names, and t_i, x_i for type and structure identifiers (with respective names t and x). The i part of identifiers is taken from a countable set of marks, in order to provide infinitely many identifiers with a given name. The general shape of paths is now $x_i.y \ldots t$, referring to the component named t of ... of the component named y of the structure bound to the identifier x_i.

In this approach, renamings are allowed to change the mark parts of identifiers, but must preserve their name parts. For instance, in the structure

$\mathtt{structure\ x}_i\ \mathtt{=\ struct\ type\ t}_j\ \mathtt{=\ int;\ ...\ end}$

we can rename $\mathtt{t}_j$ to $\mathtt{t}_k$ without changing the meaning of $\mathtt{x}_i\mathtt{.t}$, but renaming $\mathtt{t}_j$ to $\mathtt{s}_j$ would be incorrect.

This distinction between names and identifiers is required for type-checking, but can be omitted in the program source: identifiers can be recovered before type-checking by associating a mark to each binding occurrence of a name and applying the standard scoping rules to names. We will use this convention in the examples below.

Problems with the dot notation

Unlike the `open` notation, the dot notation provides a unique witness for each type component of a structure and is much closer to actual programming languages. However, it still fails to provide a reasonable notion of type identity. Consider taking a restricted view of a structure by constraining it to a smaller signature:

```
structure x = struct datatype t; val f = e; val g = e' end
structure y = (x : sig type t; val f: τ end)
```

This restriction generates a new type `y.t` incompatible with `x.t`, while in SML all views of the same structure are expected to have compatible type components. Similarly, functor application always generates new types in the result structure, even if these types are actually taken from the functor argument:

```
functor f(x: sig type t end) = struct type t = x.t end
structure x = struct datatype t end
structure y = f(x)
```

The type `y.t` is incompatible with `x.t`, even though `f` propagates the `t` component of its argument unchanged. Finally, the dot notation as presented above does not account for sharing constraints.

3.3 Manifest types

In an attempt to palliate these deficiencies of the dot notation, Harper and Lillibridge (1994) and independently the author (Leroy, 1994) have proposed to enrich signatures with type equations. The idea is to have two kinds of specifications for type components:

- *Abstract type specifications*: `type` t, which matches any implementation of t but abstracts the implementation type;
- *Manifest type specifications*: `type` $t = T$ where T is a type expression, which requires t to be implemented as a type compatible with T, and publicizes that t is compatible with T.

The motivation for this extension is that it is often needed to package a type with some operations without generating a new type, so that the operations apply to preexisting values of that type. Consider the structure

```
structure IntOrder =
  struct
    type t = int;
    fun less (x:t) (y:t) = x<y
  end
```

It is important that `IntOrder.t` remains compatible with `int`, so that `IntOrder.less` can be applied to integer values. (This behavior is in accordance with the static semantics from section 2, which assigns the same stamp to `IntOrder.t` and `int`.) More generally, no type abstraction should occur when a structure provides additional operations over an existing type, instead of defining a new type with associated operations. Manifest types in specifications answer this need: the structure `IntOrder` can be given the signature

```
sig type t = int; val less: t->t->bool end
```

from which users of the structure can deduce `IntOrder.t` = `int` and therefore apply `IntOrder.less` to integer values, just as with the stamp-based static semantics.

Manifest types also succeed in propagating type equalities through functor applications. Continuing the example above, consider a functor that extends lexicographically an ordering over a type `t` to the type `t list`.

```
functor ListOrder(Ord: sig type t; val less: t->t->bool end) =
  struct
    type t = Ord.t list
    fun less l1 l2 = ...
  end
structure IntListOrder = ListOrder(IntOrder)
```

The programmer expects `IntListOrder.t` to be compatible with `int list`. This is the result obtained with the static semantics, since the flexible stamp representing `Ord.t` is replaced by the stamp of `int` when elaborating the functor applications. With manifest types, the same result is achieved by giving `ListOrder` the signature

```
functor (Ord: sig type t; val less: t->t->bool end)
    sig type t = Ord.t list; val less: t->t->bool end
```

This is a dependent functor type: the signature of the functor result mentions the name of the functor parameter. When typing the functor application `ListOrder(IntOrder)`, the parameter is replaced by the actual argument in the result signature (following the standard elimination rule for dependent function types), resulting in

```
IntListOrder :
    sig type t = IntOrder.t list; val less: t->t->bool end
```

From this signature and the signature of `IntOrder`, it immediately follows that `IntListOrder.t` is compatible with `int list`.

The propagation of type equalities can also be ensured via a completely different approach based on strong sums instead of manifest types (MacQueen, 1986), but strong sums raise serious theoretical and practical difficulties (Harper and Mitchell, 1993; Harper et al., 1990) which are avoided in the "manifest types" approach.

In addition to recording and propagating type equalities, manifest types also palliate the other deficiencies of the "dot notation" approach described above. Taking a restricted view of a structure while preserving type compatibility can now be done as follows:

```
structure x = struct datatype t; val f = e; val g = e′ end
structure y = (x : sig type t = x.t; val f: τ end)
```

An unusual typing rule, described below in section 4, ensures that `x` satisfies the signature constraint above.

Finally, manifest types in functor argument position express sharing constraints: the functor

```
functor f(x: sig type t; ... end
          y: sig type t=x.t; ... end) = ...
```

behaves exactly like the following SML functor with a sharing constraint

```
functor f(x: sig type t; ... end
          y: sig type t; ... end
          sharing x.t = y.t) = ...
```

That is, the typing rules guarantee that the functor can only be applied to structures `x` and `y` for which we can prove that the type components `x.t` and `y.t` are the same type. Moreover, `x.t` and `y.t` are assumed to be compatible when typing the body of the functor. Therefore, the manifest type in the signature of the `y` parameter has exactly the same effects as the constraint `sharing x.t = y.t`.

Syntactic restrictions

The dot notation combined with manifest type specification therefore appears as a promising approach to type generativity and sharing. Due to the rather strong syntactic restrictions it requires, it is not clear, however, that it offers the same expressiveness as the stamp-based static semantics.

First restriction: to account for the propagation of types through functors, we had to relax the restriction that the functor parameter must not occur in the result signature, and allow dependent functor types, as in

```
f : functor (x: sig type t end) sig type t = x.t end
```

Dependent functor types alone cause no difficulties; what is problematic is their combination with the dot notation. Consider the application `f(g(y))`. If we blindly substitute the actual argument for the formal parameter in the result signature of `f`, we get

```
f(g(y)) : sig type t = g(y).t end
```

which is not a well-formed signature (it violates the restriction of the dot notation to paths). Functors with dependent types can therefore be applied only to paths, but not to arbitrary structure expressions.

Second, some syntactic restrictions also apply to sharing constraints. The sharing constraints expressible by manifest types in contravariant position are of the format `type` t; `sharing` $t = p$. These constraints are local (the sharing constraint appears next to the type declaration) and asymmetrical (one side of the constraint must be an identifier). SML's sharing constraints are more general: they are symmetrical (the two sides are paths) and not necessarily local.

In the remainder of this paper, we will prove that these syntactic restrictions can be circumvented in a systematic way (section 5) and do not prevent this approach from having the same expressiveness as the static semantics (section 6).

4 A path-based type system

Based on the discussion in the previous section, we now give a purely syntactic type system for a variant TypModL$'$ of the module calculus TypModL introduced in section 2. The main differences are the use of manifest types instead of sharing constraints in signatures, the syntactic restrictions outlined in section 3.3, and the addition of higher-order functors, that is, functors as first-class structure expressions. Higher-order functors fit easily in the type system, and simplify the calculus by obviating the distinction between definitions and programs.

Structure expressions:

$s ::= p$	access to a structure
$\mid$ $\texttt{struct}\ d\ \texttt{end}$	structure construction
$\mid$ $\texttt{functor}(x_i : S)s$	functor abstraction
$\mid$ $s(p)$	functor application

Structure body:

$$d ::= \varepsilon \mid c; d$$

Definitions:

c ::=	`type` $t_i = T$	type binding (non generative)
	$\mid$ `datatype` t_i	type creation (generative)
	$\mid$ `structure` $x_i = s$	structure binding
	$\mid$ `open` s	structure inclusion

Signature expressions:

S ::=	`sig` D `end`	simple signature
	$\mid$ `functor`$(x_i : S_1)S_2$	functor signature

Signature body:

$D ::= \varepsilon \mid C; D$

Specifications:

C ::=	`type` t_i	opaque type specification
	$\mid$ `type` $t_i = T$	manifest type specification
	$\mid$ `structure` $x_i : S$	structure specification

Type expressions:

$T ::= t_i \mid p.t \mid T_1 \to T_2$

Structure paths:

$p ::= x_i \mid p.x$

Typing environments:

$E ::= \varepsilon \mid E; C$

As explained in section 3.2, x and t stand for structure and type names, while x_i and t_i are structure and type identifiers, with the i component taken from some countable set of symbols. All identifiers bound in a given signature are required to have different names, so that access by name is unambiguous. (Typing environments E, being accessed by identifier, have no such constraint.)

We write $BV(D)$ for the set of identifiers bound (declared) by the specification D. This set is formally defined as follows:

$$\begin{aligned} BV(\varepsilon) &= \emptyset \\ BV(\texttt{type } t_i;\ D) &= \{t_i\} \cup BV(D) \\ BV(\texttt{type } t_i = T;\ D) &= \{t_i\} \cup BV(D) \\ BV(\texttt{structure } x_i : S;\ D) &= \{x_i\} \cup BV(D) \end{aligned}$$

The typing rules for this calculus are given in figures 2 and 3. Rules 19–28 define the familiar typing judgement "under assumptions E, the structure expression s has signature S", written $E \vdash s : S$. We briefly explain the most unusual rules.

The typing rule for structure identifiers (rule 19) transforms the signature S associated with the structure identifier x_i in the typing environment Γ to record the fact that abstract type components of the structure x_i are not arbitrary types, but actually come from the structure x_i. For instance, if x_i has signature `sig type` t_j `end`, then the t_j component is equal to $x_i.t$, and therefore x_i can also be assigned the signature `sig type` $t_j = x_i.t$ `end`. This operation is captured by the signature strengthening operation, written S/p, where p is a path and S is a signature that

Typing of module expressions and definitions:

$$E_1;\ \texttt{structure}\ x_i : S;\ E_2 \vdash x_i : S/x_i \quad (19)$$

$$\frac{E \vdash p : \texttt{sig}\ D_1;\ \texttt{structure}\ x_i : S;\ D_2\ \texttt{end}}{E \vdash p.x : S\{t_j \leftarrow p.t, x_k \leftarrow p.x \mid t_j \in BV(D_1), x_k \in BV(D_1)\}} \quad (20)$$

$$\frac{E \vdash S\ \texttt{signature} \qquad x_i \notin BV(E) \qquad E;\ \texttt{structure}\ x_i : S \vdash s : S'}{E \vdash \texttt{functor}(x_i : S)s : \texttt{functor}(x_i : S)S'} \quad (21)$$

$$\frac{E \vdash s : \texttt{functor}(x_i : S')S \qquad E \vdash p : S'' \qquad E \vdash S'' <: S'}{E \vdash s(p) : S\{x_i \leftarrow p\}} \quad (22)$$

$$\frac{E \vdash d : D}{E \vdash \texttt{struct}\ d\ \texttt{end} : \texttt{sig}\ D\ \texttt{end}} \quad (23) \qquad\qquad E \vdash \varepsilon : \varepsilon \quad (24)$$

$$\frac{E \vdash T\ \texttt{type} \qquad t_i \notin BV(E) \qquad E;\ \texttt{type}\ t_i = T \vdash d : D}{E \vdash (\texttt{type}\ t_i = T;\ d) : (\texttt{type}\ t_i = T;\ D)} \quad (25)$$

$$\frac{t_i \notin BV(E) \qquad E;\ \texttt{type}\ t_i \vdash d : D}{E \vdash (\texttt{datatype}\ t_i;\ d) : (\texttt{type}\ t_i;\ D)} \quad (26)$$

$$\frac{E \vdash s : S \qquad x_i \notin BV(E) \qquad E;\ \texttt{structure}\ x_i : S \vdash d : D}{E \vdash (\texttt{structure}\ x_i = s;\ d) : (\texttt{structure}\ x_i : S;\ D)} \quad (27)$$

$$\frac{E \vdash s : \texttt{sig}\ D_1\ \texttt{end} \qquad BV(D_1) \cap BV(E) = \emptyset \qquad E; D_1 \vdash d : D_2}{E \vdash (\texttt{open}\ s; d) : (D_1; D_2)} \quad (28)$$

Module subtyping:

$$\frac{E \vdash S_2 <: S_1 \qquad E;\ \texttt{structure}\ x_i : S_2 \vdash S_1' <: S_2'}{E \vdash \texttt{functor}(x_i : S_1)S_1' <: \texttt{functor}(x_i : S_2)S_2'} \quad (29)$$

$$\frac{\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \qquad E; C_1; \ldots; C_n \vdash C_{\sigma(i)} <: C_i' \text{ for } i \in \{1, \ldots, m\}}{E \vdash \texttt{sig}\ C_1; \ldots; C_n\ \texttt{end} <: \texttt{sig}\ C_1'; \ldots; C_m'\ \texttt{end}} \quad (30)$$

$$E \vdash (\texttt{type}\ t_i) <: (\texttt{type}\ t_i) \quad (31) \qquad\qquad E \vdash (\texttt{type}\ t_i = T) <: (\texttt{type}\ t_i) \quad (32)$$

$$\frac{E \vdash t_i \approx T}{E \vdash (\texttt{type}\ t_i) <: (\texttt{type}\ t_i = T)} \quad (33) \qquad \frac{E \vdash T_1 \approx T_2}{E \vdash (\texttt{type}\ t_i = T_1) <: (\texttt{type}\ t_i = T_2)} \quad (34)$$

$$\frac{E \vdash S_1 <: S_2}{E \vdash (\texttt{structure}\ x_i : S_1) <: (\texttt{structure}\ x_i : S_2)} \quad (35)$$

Type equivalence: (rules for congruence, reflexivity, symmetry, transitivity omitted)

$$E_1;\ \texttt{type}\ t_i = T;\ E_2 \vdash t_i \approx T \quad (36)$$

$$\frac{E \vdash p : \texttt{sig}\ D_1;\ \texttt{type}\ t_i = T;\ D_2\ \texttt{end}}{E \vdash p.t \approx T\{t_j \leftarrow p.t, x_k \leftarrow p.x \mid t_j \in BV(D_1), x_k \in BV(D_1)\}} \quad (37)$$

Fig. 2. Type system with paths

Well-formedness of signatures ($E \vdash S$ `signature`):

$$\frac{E \vdash D \texttt{ decl}}{E \vdash (\texttt{sig } D \texttt{ end}) \texttt{ signature}} \quad (38)$$

$$\frac{E \vdash S_1 \texttt{ signature} \qquad E; \texttt{structure } x_i : S_1 \vdash S_2 \texttt{ signature}}{E \vdash (\texttt{functor}(x_i : S_1)S_2) \texttt{ signature}} \quad (39)$$

$$E \vdash \varepsilon \texttt{ decl} \quad (40)$$

$$\frac{E \vdash T \texttt{ type} \qquad E; \texttt{type } t_i = T \vdash D \texttt{ decl}}{E \vdash (\texttt{type } t_i = T;\ D) \texttt{ decl}} \quad (41) \qquad \frac{E; \texttt{type } t_i \vdash D \texttt{ decl}}{E \vdash (\texttt{type } t_i;\ D) \texttt{ decl}} \quad (42)$$

$$\frac{E \vdash S \texttt{ signature} \qquad E; \texttt{structure } x_i : S \vdash D \texttt{ decl}}{E \vdash (\texttt{structure } x_i : S;\ D) \texttt{ decl}} \quad (43)$$

Well-formedness of types ($E \vdash T$ `type`):

$$E_1; \texttt{type } t_i;\ E_2 \vdash t_i \texttt{ type} \quad (44) \qquad E_1; \texttt{type } t_i = T;\ E_2 \vdash t_i \texttt{ type} \quad (45)$$

$$\frac{E \vdash p : \texttt{sig } D_1; \texttt{ type } t_i;\ D_2 \texttt{ end}}{E \vdash p.t \texttt{ type}} \quad (46)$$

$$\frac{E \vdash p : \texttt{sig } D_1; \texttt{ type } t_i = T;\ D_2 \texttt{ end}}{E \vdash p.t \texttt{ type}} \quad (47) \qquad \frac{E \vdash T_1 \texttt{ type} \qquad E \vdash T_2 \texttt{ type}}{E \vdash (T_1 \rightarrow T_2) \texttt{ type}} \quad (48)$$

Fig. 3. Well-formedness judgements for the type system

can be assigned to p:

$$\begin{aligned}
(\texttt{sig } D \texttt{ end})/p &= \texttt{sig } D/p \texttt{ end} \\
(\texttt{functor}(x_i : S_1)S_2)/p &= \texttt{functor}(x_i : S_1)S_2 \\
\varepsilon/p &= \varepsilon \\
(\texttt{type } t_i;\ D)/p &= \texttt{type } t_i = p.t;\ D/p \\
(\texttt{type } t_i = T;\ D)/p &= \texttt{type } t_i = T;\ D/p \\
(\texttt{structure } x_i : S;\ D)/p &= \texttt{structure } x_i : S/p.x;\ D/p
\end{aligned}$$

The signature strengthening step is essential to prove that some sharing constraints are satisfied, and also to preserve type compatibility between views of a structure. As an example of the latter, consider:

```
structure x = struct datatype t; ... end;
structure y = (x : sig type t=x.t end);
```

The manifest type in the signature of `y` is needed to ensure that `x.t` and `y.t` remain compatible. But the signature recorded for `x` in the environment, `sig type t; ... end`, is not a subtype of the signature given for `y`. Strengthening is required to give `x` the signature `sig type t=x.t; ... end`.

As an example of sharing constraint where strengthening is crucial, consider

```
structure x = struct datatype t end
structure y = struct type t = x.t end
structure f =
  functor (a: sig type t end) functor (b: sig type t = a.t end) ...
structure r = f(y)(x)
```

When typing the application `f(y)(x)`, we need to show that the signature of `x` is a subtype of the expected signature for the second argument, that is, `sig type t = y.t end` (after replacement of the first parameter `a` by the first argument `y`). This is not the case for the "natural" signature of `x`, `sig type t end`. However, by strengthening, `x` is given signature `sig type t = x.t end`, which is a subtype of `sig type t = y.t end` as expected, since we can prove the equivalence of `x.t` and `y.t` from the signature of `y`.

Since structures are dependent products, access to a structure component (rules 20 and 37) cannot return the type of the component as is, leaving dangling references to identifiers bound earlier in the structure. Instead, these identifier must be prefixed by the extraction path p, in order to preserve their meaning. For instance, if p has type

$$\texttt{sig type } t;\ \texttt{structure } x : \texttt{sig type } u = t \texttt{ end end},$$

then the signature of $p.x$ is $\texttt{sig type } u = p.t \texttt{ end}$, with $p.t$ in place of t.

Rules 38–48 check the well-formedness of a signature expression (e.g. the signature declared for a functor argument), by verifying that all structure and type identifiers appearing in the signature are bound before being used.

Rules 29–35 define a subtyping relation $<:$ between signatures: $\Gamma \vdash S <: S'$ holds if S is less general than S', that is, any structure that has signature S also has signature S'. This relation is used for functor applications (rule 22) to check that the actual argument matches the signature of the functor parameter. The subtyping relation allows two degrees of flexibility: extraneous signature components can be ignored and signature components reordered via rule 30; type equalities can be forgotten via rule 32, turning manifest types into abstract types.

The function $\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$ in rule 30 injects the components of the more general signature S' into the components of the less general signature S. The associated components must then match, as described by rules 31–35. Since two components match only if they bind identifiers with the same names, and since all identifiers in a structure have different names, σ is uniquely determined by the names of the components of the two signatures, and is injective (thus $m \leq n$ in rule 30).

Subtyping between signature components is straightforward: signatures of substructures must be in subtype relation (rule 35); manifest type specifications are included in abstract type specifications (rule 32); two manifest type specifications are included if and only if the manifest types are equivalent (rule 34); finally, an abstract type specification may be included in a manifest type specification if the type equality declared in the latter can be derived from the context (rule 33). The latter rule is needed to prove, for instance, that

$$(\texttt{sig type t; type u} = \texttt{t end}) <: (\texttt{sig type u; type t} = \texttt{u end}).$$

The signature components are compared not in the original environment E, but in E enriched with all declarations in S (the less general signature). This way, manifest type declarations in S are taken into account to establish the inclusions between signature components, even if they do not appear in the more general signature S'. For example, we have

$$(\texttt{sig type t} = \texttt{int; type u} = \texttt{t end}) <: (\texttt{sig type u} = \texttt{int end})$$

because we can prove $\texttt{t} \approx \texttt{int}$ under the assumption $\texttt{type t} = \texttt{int}$.

5 Normalization

In preparation for an equivalence result between the systems in section 2 and 4, we now show that any TypModL program can be rewritten into an equivalent program (with respect to the static semantics in section 2) that meets the syntactic restrictions imposed by TypModL$'$.

5.1 Introduction of identifiers

The first step in the rewriting is the addition of marks to identifiers in a way consistent with the scoping rules of TypModL. Introducing marks early in the normalization process allows subsequent transformations to rename identifiers in order to avoid name captures, an operation that is not always possible in the original TypModL calculus.

A simple way to perform this transformation is to add the same mark i to all occurrences of names in the original program, then consider the resulting program modulo alpha-conversion of marks. For instance, the program

```
type t = int; structure x = struct type t = bool end
```

becomes

$$\texttt{type t}_i\ \texttt{= int; structure x}_i\ \texttt{= struct type t}_i\ \texttt{= bool end}$$

which can then be alpha-converted to

$$\texttt{type t}_j\ \texttt{= int; structure x}_k\ \texttt{= struct type t}_l\ \texttt{= bool end}$$

thus distinguishing the two types `t`.

The static semantics with stamps (figure 1) easily extends to marked identifiers: signatures Σ and environments Γ are now mappings from identifiers (names + marks) to types and signatures. Since names are bound at most once in a signature, access by name in signatures is non-ambiguous: for any name y, there exists at most one mark i such that $y_i \in \mathrm{Dom}(\Sigma)$. By abuse of notation, we still write $\Sigma(y)$ for $\Sigma(y_i)$ where i is the mark such that $y_i \in \mathrm{Dom}(\Sigma)$.

The introduction of alpha-conversion in the module language actually offers an

Elaboration of structure expressions:

$$\Gamma \vdash p_s \Rightarrow \Gamma(p_s) \quad (3') \qquad \frac{\Gamma \vdash d \Rightarrow \Sigma}{\Gamma \vdash \mathtt{struct}\ d\ \mathtt{end} \Rightarrow \Sigma} \quad (4')$$

$$\frac{\Gamma \vdash s \Rightarrow \Sigma \qquad (\Sigma_1, \forall N.\, \Sigma_2) \leq \Gamma(f) \qquad \Sigma \succ \Sigma_1 \qquad N \cap (FS(\Gamma) \cup FS(\Sigma)) = \emptyset}{\Gamma \vdash f(s) \Rightarrow \Sigma_2} \quad (5')$$

$$\Gamma \vdash \varepsilon \Rightarrow \{\} \quad (6') \qquad \frac{\Gamma \vdash T \Rightarrow \tau \qquad t_i \notin \mathrm{Dom}(\Gamma) \qquad \Gamma + \{t_i \mapsto \tau\} \vdash d \Rightarrow \Sigma}{\Gamma \vdash (\mathtt{type}\ t_i = T;\ d) \Rightarrow \{t_i \mapsto \tau\} + \Sigma} \quad (7')$$

$$\frac{n \notin FS(\Gamma) \qquad t_i \notin \mathrm{Dom}(\Gamma) \qquad \Gamma + \{t_i \mapsto n\} \vdash d \Rightarrow \Sigma}{\Gamma \vdash (\mathtt{datatype}\ t_i;\ d) \Rightarrow \{t_i \mapsto n\} + \Sigma} \quad (8')$$

$$\frac{\Gamma \vdash s \Rightarrow \Sigma_1 \qquad x_i \notin \mathrm{Dom}(\Gamma) \qquad \Gamma + \{x_i \mapsto \Sigma_1\} \vdash d \Rightarrow \Sigma_2}{\Gamma \vdash (\mathtt{structure}\ x_i = s;\ d) \Rightarrow \{x_i \mapsto \Sigma_1\} + \Sigma_2} \quad (9')$$

$$\frac{\Gamma \vdash s \Rightarrow \Sigma_1 \qquad \mathrm{Dom}(\Sigma_1) \cap \mathrm{Dom}(\Gamma) = \emptyset \qquad \Gamma + \Sigma_1 \vdash d \Rightarrow \Sigma_2}{\Gamma \vdash (\mathtt{open}\ s;\ d) \Rightarrow \Sigma_1 + \Sigma_2} \quad (10')$$

Elaboration of programs:

$$\Gamma \vdash \varepsilon \Rightarrow \mathtt{ok} \quad (16') \qquad \frac{\Gamma \vdash s \Rightarrow \Sigma \qquad x_i \notin \mathrm{Dom}(\Gamma) \qquad \Gamma + \{x_i \mapsto \Sigma\} \vdash m \Rightarrow \mathtt{ok}}{\Gamma \vdash (\mathtt{structure}\ x_i = s;\ m) \Rightarrow \mathtt{ok}} \quad (17')$$

$$\frac{\begin{array}{c}\Gamma \vdash S \Rightarrow \Sigma_1 \qquad \Sigma_1 \text{ is principal for } S \text{ in } \Gamma \qquad N_1 = FS(\Sigma_1) \setminus FS(\Gamma) \\ x_i \notin \mathrm{Dom}(\Gamma) \qquad \Gamma + \{x_i \mapsto \Sigma_1\} \vdash s \Rightarrow \Sigma_2 \qquad N_2 = FS(\Sigma_2) \setminus FS(\Sigma_1) \setminus FS(\Gamma) \\ f_j \notin \mathrm{Dom}(\Gamma) \qquad \Gamma + \{f_j \mapsto \forall N_1.\, (\Sigma_1, \forall N_2.\, \Sigma_2)\} \vdash m \Rightarrow \mathtt{ok}\end{array}}{\Gamma \vdash (\mathtt{functor}\ f_j(x_i : S) = s;\ m) \Rightarrow \mathtt{ok}} \quad (18')$$

Fig. 4. Modified static semantics with identifiers and no rebindings

opportunity to simplify the elaboration rules, by making the W component (the set of stamps already in use) redundant with the elaboration environment Γ: we can now require that identifiers added to Γ are not already bound in Γ (this requirement can always be satisfied by prior alpha-conversion). In this case, the stamps already used in the program are exactly the stamps free in Γ: starting with $\Gamma = \{\}$ and $W = \emptyset$, we have $W = FS(\Gamma)$ at all elaboration steps. (This property does not hold in general if arbitrary rebindings are allowed (Russo, 1995). Consider:

```
structure x = struct datatype t end;
structure y = struct structure x = struct end; datatype u end
```

When elaborating `datatype u`, the environment is $\{\mathtt{x} \mapsto \{\}\}$, which does not contain the stamp assigned to the datatype `t`.)

The modified elaboration rules, with restricted rebindings and W replaced by $FS(\Gamma)$, are shown in figure 4. These rules are equivalent to the original elaboration rules in the following sense: if m is a program and m' the same program after introduction of marks on identifiers, $\{\}, \emptyset \vdash m \Rightarrow \mathtt{ok}$ (with the original elaboration rules from figure 1) if and only if $\{\} \vdash m' \Rightarrow \texttt{ok}$ (with the modified elaboration rules from figure 4). In the remainder of this paper, we will always use the modified elaboration rules.

5.2 Normalization of functor applications

In TypModL$'$, functor arguments are restricted to paths: $f(p)$, where p is a path, is correct, but $f(g(p))$ is prohibited. In the latter case, an intermediate binding of $g(p)$ to a structure identifier must be introduced. This transformation is expressed by the following rewrite rules over structure expressions and programs:

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{struct}\ d_1;\ \texttt{structure}\ x_i = f_k(s);\ d_2\ \texttt{end}} \\
\stackrel{a}{\rightarrow} & \texttt{struct}\ d_1;\ \texttt{structure}\ y_j = s;\ \texttt{structure}\ x_i = f_k(y_j);\ d_2\ \texttt{end} \\
 & \text{if } s \text{ is not a path and } y_j \in N \\
\multicolumn{2}{l}{\texttt{struct}\ d_1;\ \texttt{open}\ f_k(s);\ d_2\ \texttt{end}} \\
\stackrel{a}{\rightarrow} & \texttt{struct}\ d_1;\ \texttt{structure}\ y_j = s;\ \texttt{open}\ f_k(y_j);\ d_2\ \texttt{end} \\
 & \text{if } s \text{ is not a path and } y_j \in N \\
\multicolumn{2}{l}{\texttt{structure}\ x_i = f_k(s);\ m} \\
\stackrel{a}{\rightarrow} & \texttt{structure}\ y_j = s;\ \texttt{structure}\ x_i = f_k(y_j);\ m \\
 & \text{if } s \text{ is not a path and } y_j \in N \\
\multicolumn{2}{l}{\texttt{functor}\ f_i(x_j : S) = g_k(s);\ m} \\
\stackrel{a}{\rightarrow} & \texttt{functor}\ f_i(x_j : S) = (\texttt{struct structure}\ y_l = s;\ \texttt{open}\ g_k(y_l)\ \texttt{end});\ m \\
 & \text{if } s \text{ is not a path}
\end{array}$$

In the rules above, N stands for a countable set of “fresh” identifiers, disjoint from the set of identifiers used in the original program. Moreover, all identifiers in N are assumed to have names distinct from the names of the program identifiers. These two conditions ensure that the introduction of intermediate bindings does not cause name clashes.

The rewrite rules are obviously normalizing, and any program in normal form with respect to these rules contains only applications of functors to paths. It is easy to show that this transformation preserves the meaning of the program according to the static semantics:

Proposition 1

1. If $\Gamma \vdash s \Rightarrow \Sigma$ and $s \stackrel{a}{\rightarrow} s'$, then there exists a signature $\Sigma' \succ \Sigma$ such that $\Gamma \vdash s' \Rightarrow \Sigma'$.
2. If $\Gamma \vdash m \Rightarrow \texttt{ok}$ and $m \stackrel{a}{\rightarrow} m'$, then $\Gamma \vdash m' \Rightarrow \texttt{ok}$.

This transformation is an instance of Sabry and Felleisen’s A-normalization (Sabry and Felleisen, 1992), which also consists in naming the results of all function and primitive applications, but works in the more general setting of call-by-value λ-calculus. A-normalization has been developed as an alternative to continuation-passing style for compiler optimizations; it finds here an unexpected application in the area of type systems. This introduction of new names for each functor application can be viewed as the syntactic counterpart of the creation of new stamps for the generative components of functor results.

5.3 Normalization of sharing constraints

We now turn to the other syntactic restriction of the type system in section 4: it can only express sharing constraints of the form `type` t_i; `sharing` $t_i = p$ or `type` t_i; `sharing` $p = t_i$. We call these sharing constraints (where the constraint occurs next to the declaration of one of the constrained types) *local*.

The following rewrite rules over signature bodies transform arbitrary sharing constraints into local sharing constraints, by moving constraints towards the left until they hit the declaration of one of the types involved in the constraints.

$$\texttt{sharing}\ p = p;\ D \xrightarrow{s} D \qquad (1)$$

$$\texttt{type}\ t_i;\ \texttt{sharing}\ p = q;\ D \qquad (2)$$
$$\xrightarrow{s} \texttt{sharing}\ p = q;\ \texttt{type}\ t_i;\ D$$
$$\text{if } t_i \notin FV(p) \text{ and } t_i \notin FV(q)$$

$$\texttt{structure}\ x_i : S;\ \texttt{sharing}\ p = q;\ D \qquad (3)$$
$$\xrightarrow{s} \texttt{sharing}\ p = q;\ \texttt{structure}\ x_i : S;\ D$$
$$\text{if } x_i \notin FV(p) \text{ and } x_i \notin FV(q)$$

$$\texttt{type}\ t_i;\ \texttt{sharing}\ t_i = p;\ \texttt{sharing}\ q = r;\ D \qquad (4)$$
$$\xrightarrow{s} \texttt{sharing}\ q = r;\ \texttt{type}\ t_i;\ \texttt{sharing}\ t_i = p;\ D$$
$$\text{if } t_i \notin FV(q) \text{ and } t_i \notin FV(r)$$

$$\texttt{structure}\ x_i : \texttt{sig}\ D\ \texttt{end};\ \texttt{sharing}\ x_i.\overline{p} = x_i.\overline{q};\ D' \qquad (5)$$
$$\xrightarrow{s} \texttt{structure}\ x_i : \texttt{sig}\ D;\ \texttt{sharing}\ \overline{p}/D = \overline{q}/D\ \texttt{end};\ D'$$

$$\texttt{structure}\ x_i : \texttt{sig}\ D\ \texttt{end};\ \texttt{sharing}\ x_i.\overline{p} = q;\ D' \qquad (6)$$
$$\xrightarrow{s} \texttt{structure}\ x_i : \texttt{sig}\ D;\ \texttt{sharing}\ \overline{p}/D = q\ \texttt{end};\ D'$$
$$\text{if } x_i \notin FV(q)$$

$$\texttt{structure}\ x_i : \texttt{sig}\ \texttt{sharing}\ p = q;\ D\ \texttt{end};D' \qquad (7)$$
$$\xrightarrow{s} \texttt{sharing}\ p = q;\ \texttt{structure}\ x_i : \texttt{sig}\ D\ \texttt{end};\ D'$$

$$\texttt{type}\ t_i;\ \texttt{sharing}\ t_i = p;\ \texttt{sharing}\ t_i = q;\ D \qquad (8)$$
$$\xrightarrow{s} \texttt{sharing}\ p = q;\ \texttt{type}\ t_i;\ \texttt{sharing}\ t_i = p;\ D$$
$$\text{if } t_i \notin FV(p) \text{ and } t_i \notin FV(q)$$

To reduce the number of rules, we have considered sharing constraints as commutative and identified `sharing` $p = q$ with `sharing` $q = p$. We write $\overline{p}$, $\overline{q}$ for "path tails", that is, sequences of names; all type paths can be written either t_i or $x_i.\overline{p}$. We write $\overline{p}/D$ for the path obtained from the path tail $\overline{p}$ by completing the first name of $\overline{p}$ into an identifier according to the left context D:

$$\begin{array}{rcl} t/D & = & t_i \quad \text{if } t_i \in BV(D) \\ x.\overline{p}/D & = & x_i.\overline{p} \quad \text{if } x_i \in BV(D) \end{array}$$

If t_i or x_i are not bound in D for any i, then the completion is undefined and rules 5 and 6 do not apply.

Rules 2, 3 and 4 exchange a sharing constraint with the preceding signature item if this item does not define any type involved in the constraint. To prevent infinite reduction sequences, rule 4 does not allow permuting arbitrary independent sharing constraints; instead, the rule applies only if the leftmost constraint is local. Rules 5 and 6 shorten the paths involved in a constraint by moving the constraint inside the sub-signature that corresponds to the path. Rule 7 handles sharing constraints that occur in a sub-signature but do not actually depend on the sub-signature. Finally, rule 8 simplifies multiple sharing constraints over the same simple type t_i.

Proposition 2
The rewriting system $\overset{s}{\to}$ is strongly normalizing.

Proof
Consider the positions of the non-local sharing constraints in the signature expression being rewritten. Each reduction rules moves one or several non-local constraint one step to the left and does not change the positions of the other non-local constraints. □

The normalization of a signature expression does not affect the outcome of its elaboration:

Proposition 3
Assume $D \overset{s}{\to} D'$.

1. If $\Gamma \vdash D \Rightarrow \Sigma$, then $\Gamma \vdash D' \Rightarrow \Sigma$.
2. If D elaborates in Γ and $\Gamma \vdash D' \Rightarrow \Sigma$, then $\Gamma \vdash D \Rightarrow \Sigma$.

Consequently, if $S \overset{s}{\to} S'$ and S elaborates in Γ, then S and S' admit the same principal signature in Γ.

Proof
By case analysis on the reduction rules, and examination of the elaboration derivations. We prove (1) in the case of rule 6; the other cases are similar. Assume

$$\Gamma \vdash \texttt{structure}\ x_i : \texttt{sig}\ D_1\ \texttt{end};\ \texttt{sharing}\ x.\overline{p} = q;\ D_2 \Rightarrow \Sigma$$

Given the elaboration rules, the elaboration derivation is:

$$\dfrac{\Gamma \vdash D_1 \Rightarrow \Sigma_1 \qquad \dfrac{\begin{array}{c}(\Gamma + \{x_i \mapsto \Sigma_1\})(x_i.\overline{p}) = (\Gamma + \{x_i \mapsto \Sigma_1\})(q) \\ \Gamma + \{x_i \mapsto \Sigma_1\} \vdash D_2 \Rightarrow \Sigma_2\end{array}}{\Gamma + \{x_i \mapsto \Sigma_1\} \vdash \texttt{sharing}\ x_i.\overline{p} = q;\ D_2 \Rightarrow \Sigma_2}}{\Gamma \vdash \texttt{structure}\ x_i : \texttt{sig}\ D_1\ \texttt{end};\ \texttt{sharing}\ x_i.\overline{p} = q;\ D_2 \Rightarrow \{x_i \mapsto \Sigma_1\} + \Sigma_2}$$

and we have $\Sigma = \{x_i \mapsto \Sigma_1\} + \Sigma_2$. By definition of the completion operation and of the image of a path by a signature,

$$(\Gamma + \{x_i \mapsto \Sigma_1\})(x_i.\overline{p}) = \Sigma_1(\overline{p}) = \Sigma_1(\overline{p}/D_1) = (\Gamma + \Sigma_1)(\overline{p}/D_1).$$

(The completion $\overline{p}/D_1$ is defined because the type path $x_i.\overline{p}$ elaborates and x_i : `sig` D_1 `end`.) Moreover, x_i is not free in q, and we can assume $x_i \notin BV(D_1)$ after renaming of identifiers bound in D_1. Therefore,

$$(\Gamma + \{x_i \mapsto \Sigma_1\})(q) = \Gamma(q) = (\Gamma + \Sigma_1)(q).$$

It follows that $(\Gamma + \Sigma_1)(\overline{p}/D_1) = (\Gamma + \Sigma_1)(q)$. From this equality and the derivation of $\Gamma \vdash D_1 \Rightarrow \Sigma_1$, we can construct a derivation of $\Gamma \vdash D_1;$ `sharing` $\overline{p}/D_1 = q \Rightarrow \Sigma_1$ and conclude

$$\Gamma \vdash \texttt{structure}\ x_i : \texttt{sig}\ D_1;\ \texttt{sharing}\ \overline{p}/D_1 = q\ \texttt{end};\ D_2 \Rightarrow \{x_i \mapsto \Sigma_1\} + \Sigma_2$$

which is the expected result. □

We now show that if a signature expression is in normal form, then all sharing constraints contained in the signature are either local (`type` t_i; `sharing` $t_i = p$) or equate two "rigid" paths, that is, two paths referring to types bound outside the signature. Here is an example of the latter case:

```
structure S = struct datatype t; ... end
structure R = struct type t = S.t; ... end
functor(X: sig ... sharing S.t = R.t ... end) = ...
```

A sharing constraint between rigid paths in a signature is satisfied independently of the structure matched against this signature: either the two types are different and then the signature does not elaborate at all, or the two types are identical (as in the example above) and then the sharing constraint is always satisfied and can be deleted without changing the result of the elaboration. Hence, if a signature expression elaborates, we can assume that it contains no sharing constraints between two rigid paths.

Proposition 4
Let S be a signature expression that elaborates (there exists Γ and Σ such that $\Gamma \vdash S \Rightarrow \Sigma$). If S contains no rigid sharing constraints and is in normal form with respect to $\xrightarrow{s}$, then all sharing constraints in S are local.

Proof
We show that if S contains a non-local sharing constraint, then either this constraint is rigid or S is not in normal form. Let `sharing` $p = q$ be the leftmost non-local sharing constraint in S. Assume $p \neq q$ (otherwise, rule 1 applies). Let S' be the smallest sub-signature of S that contains the constraint. Consider the signature item that precedes the constraint in S'.

If $S' =$ `sig sharing` $p = q$; ... `end`, then either S' is a proper sub-signature of S and rule 7 apply, or $S' = S$ and then p and q are rigid paths (otherwise S would not elaborate).

If $S' =$ `sig` $\ldots$; `type` t_i; `sharing` $p = q$; $\ldots$ `end`, then t_i is not free in p nor q (otherwise, the constraint would be local), hence rule 2 applies.

If $S' =$ `sig` $\ldots$; `sharing` $p' = q'$; `sharing` $p = q$; $\ldots$ `end`, then the constraint $p' = q'$ is local (otherwise, $p = q$ would not be the leftmost non-local constraint in S), hence either rule 4 or rule 8 applies.

If $S' =$ `sig` $\ldots$; `structure` $x_i : S''$; `sharing` $p = q$; $\ldots$ `end`, then one of rules 3, 5 or 6 must apply. The fact that S elaborates guarantees the existence of the completions $\overline{p}/D$ and $\overline{q}/D$ in rules 5 and 6. □

6 Equivalence of the type system and the static semantics

The program expressions from the TypModL calculus of section 2, once normalized as described in the previous section, can be considered as definitions from the TypModL′ calculus in section 4, modulo the following syntactic identifications:

	Normalized TypModL		TypModL′
In signatures:	`type` t_i; `sharing` $t_i = p_t$	$\longleftrightarrow$	`type` $t_i = p_t$
In programs:	`functor` $f_i(x_j : S) = s$	$\longleftrightarrow$	`structure` $f_i =$ `functor`$(x_j : S)s$

We have therefore injected TypModL programs in normal form into the first-order fragment of TypModL′. Programs in normal form can therefore be checked at compile-time using either the static semantics from section 2 or the type system from section 4. In the remainder of this section, we prove that the two approaches give exactly the same results.

Proposition 5
Let m be a TypModL program in normal form. Then, $\{\} \vdash m \Rightarrow \texttt{ok}$ if and only if there exists a specification D such that $\varepsilon \vdash m : D$ (viewing m as a first-order TypModL′ definition).

To establish this result by induction on the derivations, we need first to set up a correspondence between the stamp-based semantic objects used for elaboration and the signature expressions used for type-checking. The idea is that the stamps in semantic objects are in one-to-one correspondence with equivalence classes of paths for the equivalence relation between types induced by the manifest type specifications in the signature expressions; this one-to-one correspondence extends to an isomorphism between the syntactic objects (type and signature expressions) and the semantic objects. For technical reasons, it is easier to define the isomorphism directly, and assign stamps to equivalence classes of paths on the fly. The translation $[T]_\Gamma$ of a type expression T in an environment Γ (mapping type identifiers to types and structure identifiers to signatures or functor signatures) is defined by:

$$\begin{aligned} [t_i]_\Gamma &= \Gamma(t_i) \\ [p_s.t]_\Gamma &= \Gamma(p_s)(t) \\ [T_1 \to T_2]_\Gamma &= [T_1]_\Gamma \to [T_2]_\Gamma \end{aligned}$$

The translation is extended to signature expressions and specifications as follows:

$$[\texttt{sig}\ D\ \texttt{end}]_\Gamma = [D]_\Gamma$$

$$
\begin{array}{rcl}
[\mathtt{functor}(x_i : S_1) S_2]_\Gamma & = & \forall N_1.\,(\Sigma_1, \forall N_2.\,\Sigma_2) \\
& & \text{where } \Sigma_1 = [S_1]_\Gamma \text{ and } \Sigma_2 = [S_2]_{\Gamma + \{x_i \mapsto \Sigma_1\}} \\
& & \text{and } N_1 = FS(\Sigma_1) \setminus FS(\Gamma) \\
& & \text{and } N_2 = FS(\Sigma_2) \setminus FS(\Sigma_1) \setminus FS(\Gamma) \\
[\varepsilon]_\Gamma & = & \{\} \\
[\mathtt{type}\ t_i;\ D]_\Gamma & = & \{t_i \mapsto n\} + [D]_{\Gamma + \{t_i \mapsto n\}} \text{ where } n \notin FS(\Gamma) \\
[\mathtt{type}\ t_i = T;\ D]_\Gamma & = & \{t_i \mapsto [T]_\Gamma\} + [D]_{\Gamma + \{t_i \mapsto [T]_\Gamma\}} \\
[\mathtt{structure}\ x_i : S;\ D]_\Gamma & = & \{x_i \mapsto [S]_\Gamma\} + [D]_{\Gamma + \{x_i \mapsto [S]_\Gamma\}}
\end{array}
$$

Finally, the translation of a typing environment E is defined as $[E] = [E]_{\{\}}$, where E is viewed as a specification in the right-hand side.

The translation defined above strongly resembles the elaboration of signature expressions in the static semantics (rules 11–15), except that it is completely deterministic: $[S]_\Gamma$ is uniquely defined up to a renaming of stamps not free in Γ. We write $=_\Gamma$ to denote equality of types and signatures up to a renaming of stamps not free in Γ. The elimination of the non-determinism introduced by rule 13 has been made possible by the transformation of sharing constraints into manifest type specifications. The following proposition relates precisely the translation $[\cdot]$ with type and signature elaboration:

Proposition 6

1. Let T be a type expression. If $[T]_\Gamma$ is defined, then $\Gamma \vdash T \Rightarrow [T]_\Gamma$. Conversely, if there exists τ such that $\Gamma \vdash T \Rightarrow \tau$, then $[T]_\Gamma$ is defined and equal to τ.
2. Let S be a normalized TypModL signature expression. If $[S]_\Gamma$ is defined, then $\Gamma \vdash S \Rightarrow [S]_\Gamma$. Conversely, if there exists Σ such that $\Gamma \vdash S \Rightarrow \Sigma$, then $[S]_\Gamma$ is defined and is the principal signature of S in Γ.

Proof

Easy structural inductions on T and S. For (2), notice that a type t_i declared by `type` t_i; `sharing` $t_i = p_t$ can only elaborate to $\Gamma(p_t)$. In all other cases, there are no sharing constraints over t_i and we obtain a principal signature by assigning a fresh stamp to t_i. □

We can now express the main inductive step for the proof of proposition 5:

Proposition 7

Let s be a normalized structure expression and d be a normalized definition. Assume defined $\Gamma = [E]$.

1. If $\Gamma \vdash s \Rightarrow \Sigma$ for some signature Σ, then there exists a signature expression S such that $E \vdash s : S$ and $[S]_\Gamma =_\Gamma \Sigma$. If $\Gamma \vdash d \Rightarrow \Sigma$ for some signature Σ, then there exists a specification D such that $E \vdash d : D$ and $[D]_\Gamma =_\Gamma \Sigma$.
2. Conversely, if $E \vdash s : S$ for some signature expression S, then $[S]_\Gamma$ is defined and $\Gamma \vdash s \Rightarrow [S]_\Gamma$. If $E \vdash d : D$ for some specification D, then $[D]_\Gamma$ is defined and $\Gamma \vdash d \Rightarrow [D]_\Gamma$.

The proof of proposition 7 makes use of three key lemmas: proposition 9 relates type equivalence and type elaboration; proposition 11 shows that the subtyping relation between signatures is equivalent to a combination of instantiation and enrichment; proposition 13 relates syntactic well-formedness and elaboration. Propositions 8, 10 and 14 are auxiliary results.

Proposition 8
Assume defined $\Gamma = [E]$. Let p be a structure path. Then, $E \vdash p : S$ for some signature expression S if and only if $\Gamma(p)$ is defined. In this case:

1. $[S]_\Gamma$ is defined and equal to $\Gamma(p)$.
2. If $S = \texttt{sig}\ D_1;\ \texttt{type}\ t_i = T;\ D_2\ \texttt{end}$, then

$$[T\{t_j \leftarrow p.t, x_k \leftarrow p.x \mid t_j \in BV(D_1), x_k \in BV(D_1)\}]_\Gamma$$

   is defined and equal to $\Gamma(p.t)$.
3. If $S = \texttt{sig}\ D_1;\ \texttt{structure}\ x_i : S';\ D_2\ \texttt{end}$, then

$$[S'\{t_j \leftarrow p.t, x_k \leftarrow p.x \mid t_j \in BV(D_1), x_k \in BV(D_1)\}]_\Gamma$$

   is defined and equal to $\Gamma(p.x)$.

Proof
The proof is by structural induction over p. (2) and (3) easily follow from (1) and the definition of $[\cdot]_\Gamma$. For (1):

Case p is a variable x_i. Then, $E \vdash x_i : S$ if and only if E is of the format $E_1;\ \texttt{structure}\ x_i : S';\ E_2$, which is equivalent to $\Gamma(x_i)$ being defined. In this case, $S = S'/x_i$ by rule 19. An easy induction on S' shows that $[S'/x_i]_\Gamma = \Gamma(x_i)$.

Case $p = q.x$. For the "if" part, assume $\Gamma(p)$ is defined. Then, so is $\Gamma(q)$. By induction hypothesis, we obtain S' such that $E \vdash q : S'$ and $[S']_\Gamma = \Gamma(q)$. Since $\Gamma(p)$ is defined, $\Gamma(q)$ contains a field named x. Hence S' is of the format $\texttt{sig}\ D_1;\ \texttt{structure}\ x_i : S'';\ D_2\ \texttt{end}$. Applying rule 20, we obtain $E \vdash p : S$ where $S = S''\{t_j \leftarrow q.t, x_k \leftarrow q.x \mid t_j \in BV(D_1), x_k \in BV(D_1)\}$. Moreover, it follows from (3) that $[S]_\Gamma = \Gamma(q.x) = \Gamma(p)$.

For the "only if" part, assume $E \vdash p : S$. Then, by rule 20, $E \vdash q : S'$ and $S' = \texttt{sig}\ D_1;\ \texttt{structure}\ x_i : S'';\ D_2\ \texttt{end}$, with $S = S''\{t_j \leftarrow q.t, x_k \leftarrow q.x \mid t_j \in BV(D_1), x_k \in BV(D_1)\}$. Applying the induction hypothesis, we get that $\Gamma(q)$ is defined and $[S']_\Gamma = \Gamma(q)$. Since S' contains a field named x, $\Gamma(p)$ is defined as well, and (3) implies $\Gamma(p) = [S]_\Gamma$. □

Proposition 9
Assume defined $\Gamma = [E]$ and $\tau_1 = [T_1]_\Gamma$ and $\tau_2 = [T_2]_\Gamma$. Then, $E \vdash T_1 \approx T_2$ if and only if $\tau_1 = \tau_2$.

Proof
The "only if" part is an easy induction on the derivation of $E \vdash T_1 \approx T_2$, using part (2) of proposition 8 in the case of rule 37.

The "if" part is more involving. To a typing environment E, we associate a rewriting relation $\stackrel{E}{\to}$ over type expressions, defined by the following axioms:

$$\begin{array}{rcl} t_i & \stackrel{E}{\to} & T \quad \text{if } E = E_1;\ \mathtt{type}\ t_i = T;\ E_2 \\ p.t & \stackrel{E}{\to} & T\{t_j \leftarrow p.t, x_k \leftarrow p.x \mid t_j \in BV(D_1), x_k \in BV(D_1)\} \\ & & \text{if } E \vdash p : \mathtt{sig}\ D_1;\ \mathtt{type}\ t_i = T;\ D_2\ \mathtt{end} \end{array}$$

This rewriting relation has the following properties (the proofs are sketched in parentheses):

1. If $T \stackrel{E}{\to} T'$, we can derive $E \vdash T \approx T'$. (Applications of the first rewriting axiom correspond to rule 36, of the second axiom to rule 37.)
2. If $T \stackrel{E}{\to} T'$ and $\tau = [T]_\Gamma$ is defined, then $\tau' = [T']_\Gamma$ is defined and $\tau = \tau'$. (Use property 1 above and the "only if" part of this proof.)
3. If $[E]$ is defined, then $\stackrel{E}{\to}$ is strongly normalizing. (Follows from the fact that since $[E]$ is defined, all manifest type specifications $\mathtt{type}\ t_i = T$ in E are such that all identifiers and paths contained in T are bound earlier than t_i in E.)
4. If T_1' and T_2' are in normal form with respect to $\stackrel{E}{\to}$, and if $[T_1']_\Gamma = [T_2']_\Gamma$, then T_1' and T_2' are syntactically identical. (Follows from the fact that a path in normal form corresponds to an abstract type specification, and therefore is assigned a stamp that is different from the stamps assigned to any other path.)

Now, assume $[T_1]_\Gamma = [T_2]_\Gamma$. Let T_1' be the normal form of T_1 with respect to $\stackrel{E}{\to}$, and T_2' be the normal form of T_2. These normal forms exists by (3). By (2) and the hypothesis $[T_1]_\Gamma = [T_2]_\Gamma$, we have $[T_1']_\Gamma = [T_2']_\Gamma$. By (4), it follows that $T_1' = T_2'$. By (1), we have derivations of $E \vdash T_1 \approx T_1'$ and $E \vdash T_2 \approx T_2'$. By symmetry and transitivity, we obtain a derivation of $E \vdash T_1 \approx T_2$. □

Proposition 10
Let Γ be an environment and Σ, Σ' be two signatures such that $\mathrm{Dom}(\Sigma) \cap \mathrm{Dom}(\Gamma) = \emptyset$ and $\mathrm{Dom}(\Sigma') \cap \mathrm{Dom}(\Gamma) = \emptyset$ and $\Sigma' \succ \varphi(\Sigma)$ for some substitution φ with $\mathrm{Dom}(\varphi) \subseteq FS(\Sigma) \setminus FS(\Gamma)$. For all signature expressions S, if $[S]_{\Gamma+\Sigma}$ is defined, then so is $[S]_{\Gamma+\Sigma'}$. Moreover, $[S]_{\Gamma+\Sigma'} =_\Gamma \varphi([S]_{\Gamma+\Sigma})$. The same result holds for a type expression T instead of a signature expression S.

Proposition 11
Let S and S' be two signature expressions containing no functor signatures. Assume defined $\Gamma = [E]$ and $\Sigma = [S]_\Gamma$ and $\Sigma' = [S']_\Gamma$. Then, $E \vdash S' <: S$ if and only if there exists a substitution φ of types for stamps such that $\mathrm{Dom}(\varphi) \subseteq FS(\Sigma) \setminus FS(\Gamma)$ and $\Sigma' \succ \varphi(\Sigma)$.

Proof
Write $S' = \mathtt{sig}\ C_1'; \ldots; C_n'\ \mathtt{end}$ and $S = \mathtt{sig}\ C_1; \ldots; C_m\ \mathtt{end}$. The proof is by structural induction over S and S'.

For the "if" part, let φ be a substitution such that $\mathrm{Dom}(\varphi) \subseteq FS(\Sigma) \setminus FS(\Gamma)$ and $\Sigma' \succ \varphi(\Sigma)$. Since $\mathrm{Dom}(\Sigma') = BV(S')$ and $\mathrm{Dom}(\varphi(\Sigma)) = \mathrm{Dom}(\Sigma) = BV(S)$, the fact that $\Sigma' \succ \varphi(\Sigma)$ implies $BV(S) \subseteq BV(S')$. It follows that there exists an injection σ from $\{1,\dots,m\}$ to $\{1,\dots,n\}$ such that for all $i \in \{1,\dots,m\}$, the signature components C_i and $C'_{\sigma(i)}$ bind the same identifier.

Let $i \in \{1,\dots,m\}$. We need to establish that $E; C'_1; \dots; C'_n \vdash C'_{\sigma(i)} <: C_i$ (1). Notice that $[E; C'_1; \dots; C'_n] = \Gamma + \Sigma'$. We prove (1) by case analysis on C_i and $C'_{\sigma(i)}$.

Case $C'_{\sigma(i)} = (\texttt{type}\ t_i)$ and $C_i = (\texttt{type}\ t_i)$. Follows from rule 31.

Case $C'_{\sigma(i)} = (\texttt{type}\ t_i = T)$ and $C_i = (\texttt{type}\ t_i)$. Follows from rule 32.

Case $C'_{\sigma(i)} = (\texttt{type}\ t_i = T')$ and $C_i = (\texttt{type}\ t_i = T)$. Since $\varphi(\Sigma(t_i)) = \Sigma'(t_i)$, we have $\varphi([T]_{\Gamma+\Sigma}) = [T']_{\Gamma+\Sigma'}$. By proposition 10, $[T]_{\Gamma+\Sigma'} = \varphi([T]_{\Gamma+\Sigma})$. Therefore, $[T]_{\Gamma+\Sigma'} = [T']_{\Gamma+\Sigma'}$. Applying proposition 9, we obtain a derivation of $E; C'_1; \dots; C'_n \vdash T \approx T'$. (1) follows from rule 34.

Case $C'_{\sigma(i)} = (\texttt{type}\ t_i)$ and $C_i = (\texttt{type}\ t_i = T)$. We show $E; C'_1; \dots; C'_n \vdash t_i \approx T$ as in the previous case and conclude (1) from rule 33.

Case $C'_{\sigma(i)} = (\texttt{structure}\ x_i : S'_x)$ and $C_i = (\texttt{structure}\ x_i : S_x)$. By construction of Σ and Σ', we have $\Sigma(x_i) = \rho([S_x]_{\Gamma+\Sigma})$ and $\Sigma'(x_i) = \rho'([S'_x]_{\Gamma+\Sigma'})$ for some renamings ρ and ρ' such that $\mathrm{Dom}(\rho) \cap FS(\Gamma+\Sigma) = \emptyset$ and $\mathrm{Dom}(\rho') \cap FS(\Gamma+\Sigma') = \emptyset$. Let ψ be the restriction of $\rho'^{-1} \circ \varphi \circ \rho$ to $FS([S_x]_{\Gamma+\Sigma'}) \setminus FS(\Gamma + \Sigma')$.

We now show that $[S'_x]_{\Gamma+\Sigma'} \succ \psi([S_x]_{\Gamma+\Sigma'})$. Applying ρ' on both sides, this amounts to showing $\Sigma'(x_i) \succ \varphi(\rho([S_x]_{\Gamma+\Sigma'}))$ (3). From the hypothesis $\Sigma' \succ \varphi(\Sigma)$, it follows that $\Sigma'(x_i) \succ \varphi(\Sigma(x_i))$. By proposition 10, $[S_x]_{\Gamma+\Sigma'} = \varphi([S_x]_{\Gamma+\Sigma}) = \varphi(\rho^{-1}(\Sigma(x_i)))$. Finally, $(\varphi \circ \rho \circ \varphi \circ \rho^{-1})(\Sigma(x_i)) = \varphi(\Sigma(x_i))$ by construction of ρ. (3) therefore holds. The expected result (1) follows from the induction hypothesis applied to S_x and S'_x with the substitution ψ and the environment $E; C'_1; \dots; C'_n$,

For the "only if" part, assume that $E \vdash S' <: S$. Let σ be the injection from $\{1,\dots,m\}$ to $\{1,\dots,n\}$ such that $E; C'_1; \dots; C'_n \vdash C'_{\sigma(i)} <: C_i$ for $i = 1,\dots,n$. The existence of σ ensures that $\mathrm{Dom}(\Sigma) \subseteq \mathrm{Dom}(\Sigma')$. We now build the required substitution φ by induction over the number m of components in S. If $m = 0$, the condition $\Sigma' \succ \varphi(\Sigma)$ holds vacuously, and we can take φ to be the identity substitution. If $m > 0$, assume the result for $m - 1$ components. Define $\Theta = [\texttt{sig}\ C_1; \dots; C_{m-1}\ \texttt{end}]_\Gamma$. Let ψ be a substitution such that $\mathrm{Dom}(\psi) \subseteq FS(\Theta) \setminus FS(\Gamma)$ and $\Sigma' \succ \psi(\Theta)$. We argue by case analysis on C_m, the last component of S.

Case $C_m = (\texttt{type}\ t_i)$. Then, $\Sigma = \Theta + \{t_i \mapsto n\}$, where n is a stamp not free in Γ nor in Θ. Take $\varphi = \psi + \{n \mapsto \Sigma'(t_i)\}$. We have

$$\mathrm{Dom}(\varphi) = \{n\} \cup \mathrm{Dom}(\psi) \subseteq \{n\} \cup FS(\Theta) \setminus FS(\Gamma) = FS(\Sigma) \setminus FS(\Gamma).$$

Moreover, $\varphi(\Theta) = \psi(\Theta)$ since n is not free in Θ, and by construction $\varphi(\Sigma(t_i)) = \Sigma'(t_i)$. Hence $\Sigma' \succ \varphi(\Sigma)$.

Case $C_m = (\texttt{type}\ t_i = T)$. Then, $\Sigma = \Theta + \{t_i \mapsto [T]_{\Gamma+\Theta}\}$. By rules 34 and 33, either t_i is declared abstract in S' and $E; C'_1; \dots; C'_n \vdash t_i \approx T$, or t_i is declared equal to some type T' in S' and $E; C'_1; \dots; C'_n \vdash T' \approx T$. In both cases, it follows from proposition 9 that $[T]_{\Gamma+\Sigma'} = \Sigma'(t_i)$. Moreover, $[T]_{\Gamma+\Sigma'} = \psi([T]_{\Gamma+\Theta})$ by proposition 10. Therefore, $\Sigma' \succ \psi(\Sigma)$, and we can take $\varphi = \psi$.

Case $C_m = (\texttt{structure}\ x_i : S_x)$. Then, $\Sigma = \Theta + \{x_i \mapsto [S_x]_{\Gamma+\Theta}\}$. By rule 35, $S'_{\sigma(m)}$ is $(\texttt{structure}\ x_i : S'_x)$ with $E; C'_1; \ldots; C'_n \vdash S'_x <: S_x$. Applying the induction hypothesis to S'_x and S_x, which are strict subterms of S' and S, we obtain a substitution ρ such that $\mathrm{Dom}(\rho) \subseteq FS([S_x]_{\Gamma+\Sigma'}) \setminus FS(\Gamma+\Sigma')$ and $[S'_x]_{\Gamma+\Sigma'} \succ \rho([S_x]_{\Gamma+\Sigma'})$.

By proposition 10, we have $[S_x]_{\Gamma+\Sigma'} = \psi([S_x]_{\Gamma+\Theta}) = \psi(\Sigma(x_i))$. Moreover, $\Sigma'(x_i) = \eta([S'_x]_{\Gamma+\Sigma'})$ for some renaming η with domain included in $FS([S'_x]_{\Gamma+\Sigma'}) \setminus FS(\Gamma+\Sigma')$. Therefore, $\Sigma'(x_i) \succ \eta(\rho(\psi(\Sigma(x_i))))$. Since $\Sigma' \succ \psi(\Theta)$, we have $FS(\psi(\Theta)) \subseteq FS(\Sigma')$. Combined with the fact that $\mathrm{Dom}(\rho) \cap FS(\Sigma') = \emptyset$ and $\mathrm{Dom}(\eta) \cap FS(\Sigma') = \emptyset$, this entails $\eta(\rho(\psi(\Theta))) = \psi(\Theta)$. It follows that $\Sigma' \succ \eta(\rho(\psi(\Sigma)))$. Finally,

$$\begin{array}{rcl}
\mathrm{Dom}(\eta\circ\rho\circ\varphi) & \subseteq & \mathrm{Dom}(\varphi) \cup \mathrm{Dom}(\rho) \cup \mathrm{Dom}(\eta) \\
& \subseteq & (FS(\Theta) \setminus FS(\Gamma)) \cup (FS([S_x]_{\Gamma+\Sigma'}) \setminus FS(\Gamma+\Sigma')) \\
& \subseteq & (FS(\Theta) \cup FS([S_x]_{\Gamma+\Sigma'})) \setminus FS(\Gamma) \;=\; FS(\Sigma) \setminus FS(\Gamma).
\end{array}$$

This shows that we can take $\varphi = \eta\circ\rho\circ\psi$ and obtain the desired result. □

Proposition 12
Assume defined $[S]_{\Gamma+\{x_i\leftarrow\Sigma\}}$. Let p be a path. If $\Gamma(p) = \varphi(\Sigma)$ for some substitution φ with $\mathrm{Dom}(\varphi) \subseteq FS(\Sigma) \setminus FS(\Gamma)$, then $[S\{x_i \leftarrow p\}]_\Gamma$ is defined and

$$[S\{x_i \leftarrow p\}]_\Gamma =_\Gamma \varphi([S]_{\Gamma+\{x_i\leftarrow\Sigma\}}).$$

Proof
We show the result for paths, then extend it to type expressions and structure expressions by structural induction. Let $\Gamma' = \Gamma + \{x_i \mapsto \Sigma\}$. Let q be a path such that $\Gamma'(q)$ is defined. Either $q = x_i.\overline{q}$, in which case $q\{x_i \leftarrow p\} = p.\overline{q}$ and $\varphi(\Gamma'(q)) = \Gamma(p.\overline{q})$ follows from the hypothesis $\varphi(\Gamma'(x_i)) = \Gamma(p)$. Or, q does not start with x_i, in which case $q\{x_i \leftarrow p\} = q$ and the constraint on $\mathrm{Dom}(\varphi)$ guarantees $\varphi(\Gamma'(q)) = \Gamma'(q)$. □

Proposition 13
Assume defined $\Gamma = [E]$. $[T]_\Gamma$ is defined if and only if $E \vdash T\ \texttt{type}$. $[S]_\Gamma$ is defined if and only if $E \vdash S\ \texttt{signature}$.

We can now prove proposition 7.

Proof
The proof of proposition 7, part (1) is by structural induction on s and d.

Case $s = p$ (elaboration rule 3'). Follows from proposition 8.

Case $s = \texttt{struct}\ d\ \texttt{end}$. By rule 4', $\Gamma \vdash d \Rightarrow \Sigma$. Applying the induction hypothesis to d, we obtain a specification D such that $E \vdash d : D$ and $[D]_\Gamma =_\Gamma \Sigma$. We conclude $E \vdash s : \texttt{sig}\ D\ \texttt{end}$ by rule 23, and $[\texttt{sig}\ D\ \texttt{end}]_\Gamma = [D]_\Gamma =_\Gamma \Sigma$ as expected.

Case $s = f_i(p)$. (Since s is part of a normalized TypModL program, the functor argument must be a path.) By rule 5', we have

$$(\Sigma_1, \forall N.\,\Sigma_2) \leq \Gamma(f_i) \quad\text{and}\quad \Gamma \vdash p \Rightarrow \Sigma_p \quad\text{and}\quad \Sigma_p \succ \Sigma_1 \quad\text{and}\quad \Sigma = \Sigma_2.$$

Since $\Gamma(f_i)$ is a functor signature, E declares f_i with a functor type $\mathtt{functor}(x_j : S_1)\ S_2$. Write $\Theta_1 = [S_1]_\Gamma$ and $\Theta_2 = [S_2]_{\Gamma+\{x_j \mapsto \Theta_1\}}$. Since $(\Sigma_1, \forall N.\, \Sigma_2) \leq [\mathtt{functor}(x_j : S_1) S_2]_\Gamma$, there exists a substitution φ such that $\mathrm{Dom}(\varphi) \subseteq FS(\Theta_1) \setminus FS(\Gamma)$ and $\Sigma_1 = \varphi(\Theta_1)$ and $\Sigma_2 = \varphi(\Theta_2)$. (We have assumed $N = FS(\Theta_2) \setminus FS(\Theta_1) \setminus FS(\Gamma)$ without loss of generality, since the condition $N \cap FS(\Gamma) = \emptyset$ is met.) Applying the induction hypothesis to $\Gamma \vdash p \Rightarrow \Sigma_p$, we get a signature expression S_p such that $E \vdash p : S_p$ and $[S_p]_\Gamma =_\Gamma \Sigma_p$. From $\Sigma_p \succ \varphi(\Theta_1)$, proposition 11 shows that $E \vdash S_p <: S_1$. We can therefore apply rule 22, obtaining $E \vdash f_i(p) : S_2\{x_i \leftarrow p\}$. By proposition 12, we get the expected result $[S_2\{x_i \leftarrow s\}]_\Gamma =_\Gamma \varphi(\Theta_2) = \Sigma$.

Case $d = \varepsilon$. Obvious.

Case $d = (\mathtt{type}\ t_i = T;\ d')$. By rule 7', $\Gamma \vdash T \Rightarrow \tau$ and $t_i \notin \mathrm{Dom}(\Gamma)$ and $\Gamma + \{t_i \mapsto \tau\} \vdash d' \Rightarrow \Sigma'$ and $\Sigma = \{t_i \mapsto \tau\} + \Sigma'$. By proposition 6, we have $\tau = [T]_\Gamma$. Therefore, $[E;\ \mathtt{type}\ t_i = T] = \Gamma + \{t_i \mapsto \tau\}$. Applying the induction hypothesis to d', we obtain D' such that $E;\ \mathtt{type}\ t_i = T \vdash d' : D'$ and $[D']_{\Gamma'} =_{\Gamma'} \Sigma'$, where Γ' is $\Gamma + \{t_i \mapsto \tau\}$. Moreover, $t_i \notin BV(E)$ follows from the hypothesis $t_i \notin \mathrm{Dom}(\Gamma)$. By rule 25, $E \vdash (\mathtt{type}\ t_i = T;\ d') : (\mathtt{type}\ t = T;\ D')$. Finally, $[\mathtt{type}\ t_i = T;\ D']_\Gamma =_\Gamma \{t_i \mapsto \tau\} + \Sigma'$, as expected.

Case $d = (\mathtt{type}\ t_i;\ d')$. Same proof as in the previous case. The condition $n \notin FS(\Gamma)$ in rule 8' ensures that $[E;\ \mathtt{type}\ t_i] =_\Gamma \Gamma + \{t_i \mapsto n\}$.

Case $d = (\mathtt{structure}\ x_i = s';\ d')$. Same proof as in the previous case.

Case $d = (\mathtt{open}\ s';\ d')$. By rule 10', $\Gamma \vdash s' \Rightarrow \Sigma_1$ and $\mathrm{Dom}(\Sigma_1) \cap \mathrm{Dom}(\Gamma) = \emptyset$ and $\Gamma + \Sigma_1 \vdash d \Rightarrow \Sigma_2$ and $\Sigma = \Sigma_1 + \Sigma_2$. By induction hypothesis, we have $E \vdash s' : S_1$ for some signature expression S_1 such that $[S_1]_\Gamma = \Sigma_1$. Therefore, $[E;\ S_1] = \Gamma + \Sigma_1$. Applying the induction hypothesis to d', we obtain D_2 such that $E;\ S_1 \vdash d' : D_2$ and $[D_2]_{\Gamma+\Sigma_1} =_{\Gamma+\Sigma_1} \Sigma_2$. By rule 28, it follows that $E \vdash (\mathtt{open}\ s';\ d') : (D_1;\ D_2)$, where we have written S_1 as $\mathtt{sig}\ D_1\ \mathtt{end}$. Moreover, $[D_1;\ D_2]_\Gamma =_\Gamma [D_1]_\Gamma + [D_2]_{\Gamma+[D_1]_\Gamma} =_\Gamma \Sigma_1 + \Sigma_2$, as expected. □

Proof

The proof of proposition 7, part (2) is by structural induction on s and d.

Case $s = p$ (typing rule 19). Follows from proposition 8 and elaboration rule 3'.

Case $s = f_i(p)$. By rule 22, we have $E \vdash f_i : \mathtt{functor}(x_i : S_1) S_2$ and $E \vdash p : S_p$ and $E \vdash S_p <: S_1$ and $S = S_2\{x_i \leftarrow p\}$.

Since f_i has signature $\mathtt{functor}(x_i : S') S$ in E, $\Gamma(f_i)$ is defined and equal to $\forall N_1.\, (\Sigma_1, \forall N_2.\, \Sigma_2)$, where $\Sigma_1 = [S_1]_\Gamma$, $\Sigma_2 = [S_2]_{\Gamma+\{x_i \mapsto \Sigma_1\}}$, $N_1 = FS(\Sigma_1) \setminus FS(\Gamma)$, and $N_2 = FS(\Sigma_2) \setminus FS(\Sigma_1) \setminus FS(\Gamma)$. By proposition 8, we have $\Gamma(p) =_\Gamma [S_p]_\Gamma$. By applying proposition 11 to the hypothesis $E \vdash S_p <: S_1$, we obtain a substitution φ such that $\Gamma(p) \succ \varphi(\Sigma_1)$ and $\mathrm{Dom}(\varphi) \subseteq N_1$. We can therefore apply the elaboration rule 5', obtaining $\Gamma \vdash f_i(p) \Rightarrow \varphi(\Sigma_2)$. Finally, $\varphi(\Sigma_2) =_\Gamma [S]_\Gamma$ by proposition 12.

Case $s = \mathtt{struct}\ d\ \mathtt{end}$ (typing rule 23). We have $S = \mathtt{sig}\ D\ \mathtt{end}$ and $E \vdash d : D$. By induction hypothesis, $\Gamma \vdash d : [D]_\Gamma$, and we conclude with rule 4'.

Case $d = \varepsilon$. Obvious.

Case $d = (\texttt{type}\ t_i = T;\ d')$. By rule 25, we have $E \vdash T\ \texttt{type}$ and $E;\ \texttt{type}\ t_i = T \vdash d' : D'$ and $D = (\texttt{type}\ t_i = T;\ D')$. By proposition 13, the hypothesis $E \vdash T\ \texttt{type}$ implies that $\tau = [T]_\Gamma$ is defined. By induction hypothesis, $\Gamma + \{t_i \mapsto \tau\} \vdash d \Rightarrow [D]_{\Gamma+\{t_i \mapsto \tau\}}$. We conclude $\Gamma \vdash (\texttt{type}\ t_i = T;\ d) : \{t_i \mapsto \tau\} + [D]_{\Gamma+\{t_i \mapsto \tau\}}$ by rule 7, which is the expected result.

Case $d = (\texttt{type}\ t_i;\ d')$, $d = (\texttt{structure}\ x_i = s;\ d')$ or $d = (\texttt{open}\ S;\ d')$. Similar to the previous case. □

Finally, proposition 5 (the main equivalence result) follows from the proposition below:

Proposition 14
Assume defined $\Gamma = [E]$. Let m be a TypModL program in normal form. Then, $\Gamma \vdash m \Rightarrow \texttt{ok}$ if and only if there exists a specification D such that $E \vdash m : D$.

Proof
The proof is by structural induction on m. The cases $m = \varepsilon$ and $m = (\texttt{structure}\ x_i = s;\ m')$ proceed as in the proof of proposition 7. We show the proof for $m = (\texttt{functor}\ f_i(x_j : S) = s;\ m')$.

Assume $\Gamma \vdash m \Rightarrow \texttt{ok}$ by rule 18. We therefore have

$$\Gamma \vdash S \Rightarrow \Sigma_1 \quad \text{and} \quad \Sigma_1 \text{ is principal for } S \text{ in } \Gamma \quad \text{and} \quad \Gamma + \{x_j \mapsto \Sigma_1\} \vdash s \Rightarrow \Sigma_2$$
$$\text{and} \quad N_1 = FS(\Sigma_1) \setminus FS(\Gamma) \quad \text{and} \quad N_2 = FS(\Sigma_2) \setminus FS(\Sigma_1) \setminus FS(\Gamma)$$
$$\text{and} \quad \Gamma + \{f_i \mapsto \forall N_1.(\Sigma_1, \forall N_2.\Sigma_2)\} \vdash m' \Rightarrow \texttt{ok}$$

By proposition 6, $[S]_\Gamma$ is defined and $\Sigma_1 =_\Gamma [S]_\Gamma$. We have $E \vdash S\ \texttt{signature}$ by proposition 13. By proposition 7, we obtain a structure S' such that $E;\ \texttt{structure}\ x_j : S \vdash s : S'$ and $[S']_{\Gamma+\{x_j \mapsto \Sigma_1\}} = \Sigma_2$. By construction of N_1 and N_2, we have $[\texttt{functor}(x_j : S)S']_\Gamma =_\Gamma \forall N_1.(\Sigma_1, \forall N_2.\Sigma_2)$. Applying the induction hypothesis to the remainder m' of the program, we obtain a specification D' such that $E;\ \texttt{structure}\ f_i = \texttt{functor}(x_j : S)s \vdash m' : D'$, hence the expected result $E \vdash m : D$ taking $D = (\texttt{structure}\ f_i = \texttt{functor}(x_j : S)S';\ D')$.

Conversely, assume $E \vdash m : D$ for some specification D. By rules 27 and 21,

$$D = (\texttt{structure}\ f_i = \texttt{functor}(x_j : S)S';\ D')$$
$$\text{and} \quad E \vdash S\ \texttt{signature} \quad \text{and} \quad E;\ \texttt{structure}\ x_j : S \vdash s : S'$$
$$\text{and} \quad E;\ \texttt{structure}\ f_i = \texttt{functor}(x_j : S)S' \vdash m' : D'.$$

Since $E \vdash S\ \texttt{signature}$, the translation $[S]_\Gamma$ is defined, by proposition 13, and is the principal signature of S, by proposition 6. Take $\Sigma_1 = [S]_\Gamma$. From $E;\ \texttt{structure}\ x_j : S \vdash s : S'$, proposition 7 proves $\Gamma + \{x_j \mapsto \Sigma_1\} \vdash s \Rightarrow \Sigma_2$ where Σ_2 is $[S']_{\Gamma+\{x_j \mapsto \Sigma_1\}}$. Applying the induction hypothesis to the typing derivation of m', we obtain $\Gamma + \{f_i \mapsto \forall N_1.(\Sigma_1, \forall N_2.\Sigma_2)\} \vdash m' \Rightarrow \texttt{ok}$, where $N_1 = FS(\Sigma_1) \setminus FS(\Gamma)$ and $N_2 = FS(\Sigma_2) \setminus FS(\Sigma_1) \setminus FS(\Gamma)$. We can therefore apply rule 18' and conclude $\Gamma \vdash m \Rightarrow \texttt{ok}$ as expected. □

7 Concluding remarks

The work presented here can be summarized as follows: just as type generativity in Modula-2 and similar languages can be described by

$$\text{type generativity} = \text{name equivalence},$$

we have formally shown that type generativity and sharing in an SML-like module calculus (with functors and multiple views) can be described as

$$\begin{aligned}&\text{type generativity and sharing}\\ &\quad = \quad \text{path equivalence} + \text{A-normalization} + \text{S-normalization}\end{aligned}$$

where "path equivalence" refers to the combination of manifest types with the dot notation, "A-normalization" is the naming of intermediate functor applications, and "S-normalization" is the flattening of sharing constraints.

Future work on this topic include extensions of the results presented here to higher-order functors and to structure generativity. It is an open problem to account for SML's structure generativity and sharing in a type system similar to the one presented here.

As for higher-order functors, neither the stamp-based static semantics nor the manifest type-based type system extend straightforwardly to higher-order functors. The simple higher-order extension of the static semantics outlined in (Milner and Tofte, 1991; Tofte, 1992) does not propagate type and structure sharing as expected. MacQueen and Tofte (1994) solve this difficulty by partially re-elaborating higher-order functors at each application, but this required a major rework of the stamp-based static semantics. The type system in section 4 has higher-order functors built in, but again not all expected type equalities are propagated through higher-order functors (see (Leroy, 1994) for examples): in the terminology of (MacQueen and Tofte, 1994), full transparency is not achieved. The author has proposed an extension of the "manifest types" mechanism that ensures full transparency for higher-order functors (Leroy, 1995), but this system does not offer the same notion of type generativity as MacQueen and Tofte's static semantics. Finding a type system for higher-order functors that is equivalent to MacQueen and Tofte's static semantics remains an open problem.

References

Aponte, M.-V. (1993). Extending record typing to type parametric modules with sharing. In *20th symposium Principles of Programming Languages*, pages 465–478. ACM Press.

Cardelli, L. (1989). Typeful programming. In Neuhold, E. J. and Paul, M., editors, *Formal description of programming concepts*, pages 431–507. Springer-Verlag.

Cardelli, L. and Leroy, X. (1990). Abstract types and the dot notation. In Broy, M. and Jones, C. B., editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland. Also available as research report 56, DEC Systems Research Center.

Cardelli, L. and MacQueen, D. B. (1988). Persistence and type abstraction. In Atkinson, M. P., Buneman, P., and Morrison, R., editors, *Data types and persistence*. Springer-Verlag.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *Computing surveys*, 17(4):471–522.

Gran, G. (1995). An error in rule 55 of the definition of Standard ML. Personal communication, June 1995.

Harper, R. and Lillibridge, M. (1994). A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*, pages 123–137. ACM Press.

Harper, R., Milner, R., and Tofte, M. (1987). A type discipline for program modules. In *TAPSOFT 87*, volume 250 of *Lecture Notes in Computer Science*, pages 308–319. Springer-Verlag.

Harper, R. and Mitchell, J. C. (1993). On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252.

Harper, R., Mitchell, J. C., and Moggi, E. (1990). Higher-order modules and the phase distinction. In *17th symposium Principles of Programming Languages*, pages 341–354. ACM Press.

Leroy, X. (1994). Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages*, pages 109–122. ACM Press.

Leroy, X. (1995). Applicative functors and fully transparent higher-order modules. In *22nd symposium Principles of Programming Languages*, pages 142–153. ACM Press.

Liskov, B. and Guttag, J. (1986). *Abstraction and specification in program development*. MIT Press.

MacQueen, D. B. (1984). Modules for Standard ML. In *Lisp and Functional Programming 1984*, pages 198–207. ACM Press.

MacQueen, D. B. (1986). Using dependent types to express modular structure. In *13th symposium Principles of Programming Languages*, pages 277–286. ACM Press.

MacQueen, D. B. and Tofte, M. (1994). A semantics for higher-order functors. In Sannella, D., editor, *Programming languages and systems – ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag.

Milner, R. and Tofte, M. (1991). *Commentary on Standard ML*. The MIT Press.

Milner, R., Tofte, M., and Harper, R. (1990). *The definition of Standard ML*. The MIT Press.

Mitchell, J. C. and Plotkin, G. D. (1988). Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502.

Russo, C. (1995). Generativity of names in the static semantics of Standard ML. Personal communication, February 1995.

Sabry, A. and Felleisen, M. (1992). Reasoning about programs in continuation-passing style. In *Lisp and Functional Programming 1992*, pages 288–298.

Tofte, M. (1988). Operational semantics and polymorphic type inference. PhD thesis CST-52-88, University of Edinburgh.

Tofte, M. (1992). Principal signatures for higher-order program modules. In *19th symposium Principles of Programming Languages*, pages 189–199. ACM Press.

Wirth, N. (1983). *Programming in Modula-2*. Springer-Verlag.