LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science — The University of Edinburgh

Standard ML Type Generativity as Existential Quantification

by

Claudio V. Russo

LFCS Report Series **ECS–LFCS–96–344**

LFCS **June 1996**

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Standard ML Type Generativity as Existential Quantification

Claudio V. Russo*

Technical Report ECS–LFCS–96–344
Department of Computer Science
University of Edinburgh

June 25, 1996

Abstract: One of the distinguishing features of Standard ML is the use of type *generativity*. Each declaration of a datatype binds a globally fresh *type name* to the type identifier introduced. Type generativity has been regarded as an extra-logical device which, though desirable in a programming language to ensure data abstraction, bears no close resemblance to type theoretic constructs. We show that it corresponds precisely to existential quantification over types, and use the observation to suggest proper extensions to the current static semantics of Standard ML.

Keywords: language design, type theory, modules, Standard ML.

1 Introduction

Standard ML[13] has a rich modules language. Core language declarations of value and type identifiers can be packaged together into possibly nested *structures*. Access to structure components is by the dot notation and provides good control of the name space in a large program development. Structures are *transparent*: the identity of type components within a structure is evident even outside the structure[10].

ML provides two forms of data abstraction. The first is the *datatype declaration* which *generates* a new type with constructors mediating between it and its concrete representation. Each defining occurrence of a datatype is associated with a unique internal stamp, or *generative* type name; two structurally equivalent declarations result in different types. Hiding access to the constructors makes the type *abstract*. The second form is the *functor*. A functor is a mapping from structures to structures and is introduced by specifying an interface, or *signature*, for the formal argument, and providing an implementation of the body. The body may mention type and value components of the argument. Leaving some

*Dept. of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, e-mail: `cvr@dcs.ed.ac.uk`. Supported by an award from the EPSRC. Thanks to D. Sannella, D. Aspinall and J. McKinna.

type identities in the interface unspecified constrains the functor to behave parametrically on all possible type arguments. A functor may be applied to any structure which supplies *at least* the components its signature requires. Datatypes defined in the functor body receive generative names each time the functor is applied; this guarantees type security.

Traditionally, generativity is explained by the use of a compile-time, global state of names which is extended each time a datatype is declared or a functor applied, reflecting the operational approach used in the Definition[13]. On the other hand, the type theoretic approach to data abstraction relies on weak existential types[15] and is state-less. The relationship between generativity and existential quantification has been alluded to in the literature but was never made precise[15, 4]. We present a variant of the Standard ML static semantics, based directly on the use of existential types, which is provably equivalent to the generative semantics. We claim that the formulation is conceptually clearer, and use it to suggest interesting extensions to Standard ML.

2 Syntax

The syntax of our modules language is defined by

$$
\begin{array}{lrcl}
\text{Structure Paths:} & sp & ::= & x \mid sp.x \\
\text{Type Expressions:} & te & ::= & t \mid sp.t \mid te \rightarrow te' \\
\text{Signature Bodies:} & D & ::= & \mathbf{type}\ t = te; D \\
 & & \mid & \mathbf{type}\ t; D \\
 & & \mid & \mathbf{structure}\ x : Sg; D \\
 & & \mid & \epsilon_D \\
\text{Signature Expressions:} & Sg & ::= & \mathbf{sig}\ D\ \mathbf{end} \\
\text{Structure Bodies:} & d & ::= & \mathbf{type}\ t = te; d \\
 & & \mid & \mathbf{datatype}\ t = te; d \\
 & & \mid & \mathbf{structure}\ x = se; d \\
 & & \mid & \mathbf{functor}\ f\ (x : Sg) = se\ \mathbf{in}\ d \\
 & & \mid & \epsilon_d \\
\text{Structure Expressions:} & se & ::= & sp \\
 & & \mid & \mathbf{struct}\ d\ \mathbf{end} \\
 & & \mid & f\ se
\end{array}
$$

where $t \in \text{TyId}$, $x \in \text{StrId}$ and $f \in \text{FunId}$ range over disjoint sets of type, structure and functor identifiers. The syntax is based on Standard ML [13], however, since we are only concerned with the effect of module expressions on core language types, we have omitted value bindings to expressions from the core language and constructor bindings in type declarations. Our results should carry over easily to deal with local and parameterised type declarations.

To ease the presentation of signature elaboration, we have replaced the *post hoc* use of type sharing specifications **sharing** $sp.t = sp'.u$ [13] by manifest type bindings **type** $t = te; D$ [7, 3]. The result is that elaboration becomes completely syntax directed and we no

$\alpha,\beta,\delta,\gamma$	$\in \text{TypeName} \stackrel{\text{def}}{=}$ an infinite, denumerable set	type names
N,M,P,Q	$\in \text{NameSet} \stackrel{\text{def}}{=} \text{Fin}(\text{TypeName})$	finite name sets
τ or α or $\tau \to \tau'$	$\in \text{Type} \stackrel{\text{def}}{=} \text{TypeName} \uplus (\text{Type} \times \text{Type})$	semantic types
S	$\in \text{Str} \stackrel{\text{def}}{=} \text{TyId} \stackrel{\text{fin}}{\to} \text{Type} \times \text{StrId} \stackrel{\text{fin}}{\to} \text{Str}$	structures
X or $\exists N.S$	$\in \text{ExStr} \stackrel{\text{def}}{=} \text{NameSet} \times \text{Str}$	existential structures
L or $\Lambda N.S$	$\in \text{Sig} \stackrel{\text{def}}{=} \text{NameSet} \times \text{Str}$	parameterised structures
Φ or $\forall N.S \to X$	$\in \text{FunSig} \stackrel{\text{def}}{=} \text{NameSet} \times (\text{Str} \times \text{ExStr})$	functor signatures
C or (C_t, C_x, C_f)	$\in \text{Context} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \text{TyId} \stackrel{\text{fin}}{\to} \text{Type} \times \\ \text{StrId} \stackrel{\text{fin}}{\to} \text{Str} \times \\ \text{FunId} \stackrel{\text{fin}}{\to} \text{FunSig} \end{array} \right.$	contexts

Figure 1: Semantic Objects

longer have to introduce the notion of *principal signature* [13]. Our simplification is not essential [9, 1].

Finally, Standard ML functors may only be declared at top-level. To avoid introducing a further phrase class, we instead allow *local* declarations of functors in structure bodies, e.g. **functor** f $(x : Sg)$ $=$ se **in** d: the scope of f is d. Moreover, f does *not* become a component of the resulting semantic structure. Structure bodies thus serve the function of ML's top-level. Note that the language is still first-order: functors may neither take functors as arguments nor return them as results.

In Standard ML, the structure expression **struct** d **end** introduces a generative *structure name*, used to identify the module. We ignore the largely orthogonal issues of structure identity and the use of structure sharing specifications within signatures, but retain the phrase for familiarity.

3 Semantic Objects

Figure 1 defines the *semantic objects* assigned to module expressions. They serve the role of types in the module semantics. We let O range over all semantic objects.

Notation. For sets A and B, $\text{Fin}(A)$ denotes the set of *finite subsets* of A, and $A \stackrel{\text{fin}}{\to} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B. A finite map will often be written explicitly as a set in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, $k \geq 0$. Let f and g be finite maps. $\text{Dom}(f)$ and $\text{Rng}(f)$ denote the domain of definition and range of f. The finite map $f + g$ has domain $\text{Dom}(f) \cup \text{Dom}(g)$ and values $(f+g)(a) \stackrel{\text{def}}{=}$ if $a \in \text{Dom}(g)$ then $g(a)$ else $f(a)$. If $\text{Rng}(g) = \text{Dom}(f)$ then $f \circ g$ is the finite map with values $(f \circ g)(a) \stackrel{\text{def}}{=} f(g(a))$. For $A \subseteq \text{Dom}(f)$, the *restriction* $f \downarrow A$ is the finite map with domain

A and values $(f \downarrow A)(a) \stackrel{\text{def}}{=} f(a)$.

Definition 1 (Semantic Structures). A semantic structure S consists of a pair (S_t, S_x) of finite maps binding type identifiers to semantic types and structure identifiers to semantic structures respectively. For notational convenience we will define extension and retrieval functions $[t = \tau]S \stackrel{\text{def}}{=} (\{t \mapsto \tau\} + S_t, S_x)$, $S(t) \stackrel{\text{def}}{=} S_t(t)$, $[x : S']S \stackrel{\text{def}}{=} (S_t, \{x \mapsto S'\} + S_x)$ and $S(x) \stackrel{\text{def}}{=} S_x(x)$. Let ϵ_S denote the empty semantic structure $(\emptyset, \emptyset)$.

Note that Λ, $\exists$ and $\forall$ bind type names. The object $\Lambda N.S$ describes a family of structures, whose members are obtained by substituting types for the parameters in N. The object $\exists N.S$ on the other hand, is a semantic structure in which occurrences of certain types have been made abstract by existential quantification. In a functor signature $\forall N.S \to X$, names in N are bound simultaneously in S and X. These names capture the type components of the argument structure S on which the functor behaves parametrically; their possible occurrence in the result X caters for the propagation of type identities from the functor's actual argument. The range X is always of the form $X \equiv \exists M.S'$ for a set M of names. Typically, M is non-empty. It corresponds to the generative name set of the functor signature as in the Definition of Standard ML [13]. Applying a functor with the above signature results in a variant of the structure S', obtained by choosing fresh names to replace those in M.

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

Definition 2 (Contexts). A context C consists of a triple (C_t, C_x, C_f) of finite maps binding type identifiers to semantic types, structure identifiers to semantic structures and functor identifiers to functor signatures respectively. We define extension and retrieval functions $C[t = \tau] \stackrel{\text{def}}{=} (C_t + \{t \mapsto \tau\}, C_x, C_f)$, $C(t) \stackrel{\text{def}}{=} C_t(t)$, $C[x : S] \stackrel{\text{def}}{=} (C_t, C_x + \{x \mapsto S\}, C_f)$, $C(x) \stackrel{\text{def}}{=} C_x(x)$, $C[f : \Phi] \stackrel{\text{def}}{=} (C_t, C_x, C_f + \{f \mapsto \Phi\})$ and $C(f) \stackrel{\text{def}}{=} C_f(f)$.

Note that rebindings to identifiers take precedence to the right in both semantic structures and contexts. However, we "extend" structures to the left, and contexts to the right.

We will let $\mathrm{FN}(O)$ denote the set of names free in O, where the notions of free and bound name are defined as usual.

Definition 3 (Renamings). A *renaming* is a finite map from type names to type names. We let ρ, σ, and π range over renamings. We will use the more suggestive notation $[M/N]$ to denote a *bijective* renaming with domain N and range M, which simply swaps names. The effect of *applying* a renaming ρ to a name α, written $\rho\langle\alpha\rangle$, is defined to be $\rho\langle\alpha\rangle \stackrel{\text{def}}{=}$ if $\alpha \in \mathrm{Dom}(\rho)$ then $\rho(\alpha)$ else α. We extend the operation of renaming free names compositionally to all semantic objects in such a way that bound variables are renamed *only* when necessary to avoid capture of names; and $\rho\langle O\rangle \equiv \sigma\langle O\rangle$ whenever $\rho\langle\alpha\rangle = \sigma\langle\alpha\rangle$ for every $\alpha \in \mathrm{FN}(O)$. Let $\mathrm{Inv}(\rho) \stackrel{\text{def}}{=} \mathrm{Dom}(\rho) \cup \mathrm{Rng}(\rho)$ describe the set of names *involved* in the renaming ρ.

$$\boxed{C \vdash sp : S} \qquad \frac{C(x) = S}{C \vdash x : S} \qquad \frac{C \vdash sp : S' \quad S'(x) = S}{C \vdash sp.x : S}$$

$$\boxed{C \vdash te \rhd \tau} \qquad \frac{C(t) = \tau}{C \vdash t \rhd \tau} \qquad \frac{C \vdash sp : S' \quad S'(t) = \tau}{C \vdash sp.t \rhd \tau} \qquad \frac{C \vdash te \rhd \tau \quad C \vdash te' \rhd \tau'}{C \vdash te \to te' \rhd \tau \to \tau'}$$

$$\boxed{C \vdash D \rhd L} \qquad \frac{C \vdash te \rhd \tau \quad C[t = \tau] \vdash D \rhd \Lambda Q.S \quad Q \cap \mathrm{FN}(\tau) = \emptyset}{C \vdash \mathbf{type}\ t \ = \ te; D \rhd \Lambda Q.[t = \tau]S}$$

$$\frac{C[t = \alpha] \vdash D \rhd \Lambda P.S \quad \alpha \notin \mathrm{FN}(C) \cup P}{C \vdash \mathbf{type}\ t; D \rhd \Lambda\{\alpha\} \cup P.[t = \alpha]S}$$

$$\frac{C \vdash Sg \rhd \Lambda P.S \quad P \cap \mathrm{FN}(C) = \emptyset \quad C[x \ : \ S] \vdash D \rhd \Lambda Q.S' \quad Q \cap (P \cup \mathrm{FN}(S)) = \emptyset}{C \vdash \mathbf{structure}\ x \ : \ Sg; D \rhd \Lambda P \cup Q.[x \ : \ S]S'}$$

$$\frac{}{C \vdash \epsilon_D \rhd \Lambda\emptyset.\epsilon_S} \qquad \frac{C \vdash D \rhd L' \quad L' \stackrel{\alpha}{\equiv} L}{C \vdash D \rhd L}$$

$$\boxed{C \vdash Sg \rhd L} \qquad \frac{C \vdash D \rhd L}{C \vdash \mathbf{sig}\ D\ \mathbf{end} \rhd L} \qquad \frac{C \vdash Sg \rhd L' \quad L' \stackrel{\alpha}{\equiv} L}{C \vdash Sg \rhd L}$$

Figure 2: Judgements common to both static semantics

Definition 4 (α-Equivalence). The notion of α-equivalence of semantic objects is defined as usual, in particular: for $X \equiv \exists P.S$ and $X' \equiv \exists P'.S'$, $X \stackrel{\alpha}{\equiv} X'$ iff there is a *bijective* renaming $[P'/P]$ such that $[P'/P]\langle S\rangle \equiv S'$ and $\mathrm{FN}(X) = \mathrm{FN}(X')$.

Definition 5 (Enrichment Relation). Given two structures $S \equiv (S_t, S_x)$ and $S' \equiv (S'_t, S'_x)$, S *enriches* S', written $S \succeq S'$, iff

1. $\mathrm{Dom}(S_t) \supseteq \mathrm{Dom}(S'_t)$ and for all $t \in \mathrm{Dom}(S'_t)$, $S(t) = S'(t)$, and
2. $\mathrm{Dom}(S_x) \supseteq \mathrm{Dom}(S'_x)$ and for all $x \in \mathrm{Dom}(S'_x)$, $S(x) \succeq S'(x)$.

Definition 6 (Instantiation Relation). $S \to X \leq \forall N.S' \to X'$ iff there exists a substitution, φ, of semantic types for type names, with $\mathrm{Dom}(\varphi) = N$, such that $S \equiv \varphi(S')$ and $X \stackrel{\alpha}{\equiv} \varphi(X')$.

4 Static Semantics

In this section we introduce two distinct static semantics for structure bodies and structure expressions. The systems rely on shared judgement forms dealing with structure paths, type expressions, signature bodies and signature expressions. The common judgements are shown in Figure 2. We can factor out these judgements because they do not generate any new *free* names in their conclusions.

$$\boxed{C, N \vdash d \Rightarrow S, M} \qquad \frac{}{C, N \vdash \epsilon_d \Rightarrow \epsilon_S, \emptyset} \text{ (E-1)}$$

$$\frac{C \vdash \mathit{te} \triangleright \tau \quad C[t = \tau], N \vdash d \Rightarrow S, M}{C, N \vdash \mathbf{type}\ t\ =\ \mathit{te}; d \Rightarrow [t = \tau]S, M} \text{ (E-2)}$$

$$\frac{C \vdash \mathit{te} \triangleright \tau \quad \delta \notin N \quad C[t = \delta], N \cup \{\delta\} \vdash d \Rightarrow S, Q}{C, N \vdash \mathbf{datatype}\ t\ =\ \mathit{te}; d \Rightarrow [t = \delta]S, \{\delta\} \cup Q} \text{ (E-3)}$$

$$\frac{C, N \vdash \mathit{se} \Rightarrow S, P \quad C[x\ :\ S], N \cup P \vdash d \Rightarrow S', Q}{C, N \vdash \mathbf{structure}\ x\ =\ \mathit{se}; d \Rightarrow [x\ :\ S]S', P \cup Q} \text{ (E-4)}$$

$$\frac{\begin{array}{ll} C \vdash \mathit{Sg} \triangleright \Lambda P.S' & P \cap N = \emptyset \\ C[x\ :\ S'], N \cup P \vdash \mathit{se} \Rightarrow S'', Q & C[f\ :\ \forall P.S' \to \exists Q.S''], N \vdash d \Rightarrow S, M \end{array}}{C, N \vdash \mathbf{functor}\ f\ (x : \mathit{Sg})\ =\ \mathit{se}\ \mathbf{in}\ d \Rightarrow S, M} \text{ (E-5)}$$

$$\boxed{C, N \vdash \mathit{se} \Rightarrow S, M} \qquad \frac{C \vdash \mathit{sp} : S}{C, N \vdash \mathit{sp} \Rightarrow S, \emptyset} \text{ (E-6)} \qquad \frac{C, N \vdash d \Rightarrow S, M}{C, N \vdash \mathbf{struct}\ d\ \mathbf{end} \Rightarrow S, M} \text{ (E-7)}$$

$$\frac{C, N \vdash \mathit{se} \Rightarrow S', P \quad S'' \to \exists Q.S \leq C(f) \quad S' \succeq S'' \quad Q \cap (N \cup P) = \emptyset}{C, N \vdash f\ \mathit{se} \Rightarrow S, P \cup Q} \text{ (E-8)}$$

Figure 3: Elaboration rules based on generative names

4.1 Elaboration Semantics

Figure 3 presents the Standard ML style static semantics for our language. Consider the form of the judgements $C, N \vdash d \Rightarrow S, M$ and $C, N \vdash \mathit{se} \Rightarrow S, M$. The explicit set of names, N, is meant to capture a superset of the names generated so far. Elaboration produces, besides the semantic object S, the set of names M generated *during* the elaboration of the phrase d or se. The name sets are threaded through elaboration trees in a global, state-like manner. This avoids any unsafe confusion of existing names with the fresh names generated by datatypes (Rule (E-3)) and functor applications (Rule (E-8)). The generative nature of elaboration is expressed by the following property:

Property 1 (Generativity). *If* $C, N \vdash d/\mathit{se} \Rightarrow S, M$ *then* $N \cap M = \emptyset$.[1]

Note that the sets of generated names are *not* redundant. Suppose we deleted them from the elaboration judgements and replaced occurrences of N by $\mathrm{FN}(C)$. Then it is easy to see that

$$\begin{array}{lll} \mathbf{structure}\ x = & \mathbf{struct} & \mathbf{datatype}\ t = \mathbf{int}\ \mathbf{end}; \\ \mathbf{structure}\ y = & \mathbf{struct} & \mathbf{structure}\ x = \mathbf{struct}\ \mathbf{end}; \\ & & \mathbf{datatype}\ u = \mathbf{bool} \\ & \mathbf{end} & \end{array} \Rightarrow \begin{array}{ll} [x: & [t = \alpha]] \\ [y: & [x\ :\ \epsilon_S] \\ & [u = \alpha]] \end{array}$$

so that $x.t = y.u$ even though it is definitely the case that $\mathbf{bool} \neq \mathbf{int}$. This is a type insecurity. The problem arises because the name α generated for t is no longer free in the

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

$$\boxed{C \vdash d : X} \qquad \frac{}{C \vdash \epsilon_d : \exists \emptyset . \epsilon_S} \text{ (T-1)}$$

$$\frac{C \vdash te \triangleright \tau \quad C[t = \tau] \vdash d : \exists P.S \quad P \cap \mathrm{FN}(\tau) = \emptyset}{C \vdash \mathbf{type}\ t\ =\ te; d : \exists P.[t = \tau]S} \text{ (T-2)}$$

$$\frac{C \vdash te \triangleright \tau \quad \delta \notin \mathrm{FN}(C) \cup Q \quad C[t = \delta] \vdash d : \exists Q.S}{C \vdash \mathbf{datatype}\ t\ =\ te; d : \exists \{\delta\} \cup Q.[t = \delta]S} \text{ (T-3)}$$

$$\frac{C \vdash se : \exists P.S \quad P \cap \mathrm{FN}(C) = \emptyset \quad C[x\ :\ S] \vdash d : \exists Q.S' \quad Q \cap (P \cup \mathrm{FN}(S)) = \emptyset}{C \vdash \mathbf{structure}\ x\ =\ se; d : \exists P \cup Q.[x\ :\ S]S'} \text{ (T-4)}$$

$$\frac{C \vdash Sg \triangleright \Lambda P.S \quad P \cap \mathrm{FN}(C) = \emptyset \quad C[x\ :\ S] \vdash se : X' \quad C[f\ :\ \forall P.S \to X'] \vdash d : X}{C \vdash \mathbf{functor}\ f\ (x : Sg)\ =\ se\ \mathbf{in}\ d : X} \text{ (T-5)}$$

$$\boxed{C \vdash se : X} \qquad \frac{C \vdash sp : S}{C \vdash sp : \exists \emptyset . S} \text{ (T-6)} \qquad \frac{C \vdash d : X}{C \vdash \mathbf{struct}\ d\ \mathbf{end} : X} \text{ (T-7)}$$

$$\frac{C \vdash se : \exists P.S' \quad P \cap (\mathrm{FN}(C(f)) \cup Q) = \emptyset \quad S'' \to \exists Q.S \le C(f) \quad S' \succeq S''}{C \vdash f\ se : \exists P \cup Q.S} \text{ (T-8)}$$

$$\boxed{\stackrel{\alpha}{\equiv}\text{-conversion}} \qquad \frac{C \vdash d : X' \quad X' \stackrel{\alpha}{\equiv} X}{C \vdash d : X} \text{ (T-9)} \qquad \frac{C \vdash se : X' \quad X' \stackrel{\alpha}{\equiv} X}{C \vdash se : X} \text{ (T-10)}$$

Figure 4: Type system based on existential quantification

context by the time we need to guess a fresh name for u: it is hiding in the shadow of the second binding to x in the context $[x\ :\ [t = \alpha]][x\ :\ \epsilon_S]$. In the elaboration semantics, the prior use of α will be recorded in the set of names $N \cup \{\alpha\}$ at the point at which we are elaborating **datatype** u = **int**, forcing us to choose a distinct name. These observations motivate:

Definition 7 (Rigidity). A context C is *rigid* w.r.t. N, written C, N rigid, iff $\mathrm{FN}(C) \subseteq N$.

As long as we start with C, N rigid, as a consequence of Property 1, those names in M resulting from the elaboration of d and se will never be confused with names visible in the context, even if these are temporarily hidden by bindings added to C during sub-elaborations.

4.2 Type Theoretic Semantics

Figure 4 presents an alternative static semantics for structure bodies and expressions, defined by the judgements $C \vdash d : \exists M.S$ and $C \vdash se : \exists M.S$. Rather than maintaining a global state of names threaded through elaborations, we adopt the view that module expressions type-check to existentially quantified semantic structures. The key idea is to replace *global* generativity with implicit elimination and introduction of existential types — in essence: *local* generativity. In the rules, the side conditions on names sets prevent capture of names in the usual way. Rules (T-9) and (T-10) ensure that we can always rename bound variables if necessary to satisfy the side conditions. We will give an intuitive explanation of some of the rules:

$\boxed{\textbf{type}\ t\ =\ te; d}$ We simplify te to find a corresponding semantic type τ. We then check d in the context extended by the binding of τ to t, to obtain an existentially quantified structure $\exists P.S$. Provided τ does not contain any of the names bound by P in $\exists P.S$ we can eliminate the existential, extend S by the binding to t and then hide the hypothetical types by existentially quantifying over the resulting structure.

$\boxed{\textbf{datatype}\ t\ =\ te; d}$ We proceed as in the previous case, except that instead of binding t to its definition, we bind it a *locally fresh* type name δ. Type checking d results in semantic structure $\exists Q.S$, which may contain occurrences of δ. We conceptually eliminate the existential quantification over S, extend S by the abstract binding to t and then existentially quantify over both the abstract type δ and the Q hypothetical types we just eliminated.

$\boxed{\textbf{structure}\ x\ =\ se; d}$ We typecheck se to an existential structure $\exists P.S$. We locally eliminate the existential, introducing fresh hypothetical types P, and type check d in the suitably extended context to obtain a semantic structure $\exists Q.S'$. Now $\exists Q.S'$ may contain some of the locally introduced hypothetical types in P, and these should not escape their scope. We eliminate the existential $\exists Q.S'$, extend the result by the binding to x and existentially quantify over the types $P \cup Q$.

$\boxed{\textbf{functor}\ f\ (x : Sg)\ =\ se\ \textbf{in}\ d}$ The signature expression Sg denotes a family of semantic structures, $\Lambda P.S$. We want f to be applicable to all enrichments of instances of $\Lambda P.S$. To this end, we type check the body se of f in a context extended with a generic instance of $\Lambda P.S$, i.e. S, where we ensure that P is a locally fresh choice of type names. Now se will typecheck to an existentially quantified structure X', which may contain occurrences of our generic names P. Since the functor typechecks for a generic choice of names, it will typecheck for any substitution of types for these names. We universally quantify over P, and add the polymorphic binding to f locally to the context. Type checking the scope d of the functor declaration yields the result X for the entire phrase. Note that the result does not contain a "functor binding" for f.

$\boxed{f\ se}$ Type check se to obtain an existentially quantified structure $\exists P.S'$. Locally eliminate the quantifier to see whether the functor may be applied to the structure (a combination of instantiation and enrichment), obtaining an existentially quantified result of type $\exists Q.S$. The functor may propagate some of the hypothetical types in P. To prevent them escaping their scope, we hide them by extending the existential quantification over S to cover both P and Q.

$\boxed{sp}$ We simply look up sp in the context to obtain an (unquantified) semantic structure. Note that the dot notation is completely independent of the treatment of existentials.

Before proceeding to the statement of the main result we need one last concept:

Definition 8 (Well-formedness). A functor signature $\Phi \equiv \forall N.S \to X$ is *well-formed*, written $\Phi\,\mathrm{WF}$, iff for every $\alpha \in N$, $\alpha \in \mathrm{FN}(X)$ only if $\alpha \in \mathrm{FN}(S)$.

A context C is well-formed, written $C\,\mathrm{WF}$, precisely when all the functor signatures in its range are well-formed.

The well-formedness of a functor signature Φ ensures that whenever we apply the functor, the free names of the result are either propagated from the actual argument, or were already free in Φ:

Lemma 2. *If* $\Phi\,\mathrm{WF}$ *and* $S \to X \leq \Phi$ *then* $\mathrm{FN}(X) \subseteq \mathrm{FN}(S) \cup \mathrm{FN}(\Phi)$.

In the elaboration semantics, if we start with a well-formed context, then the names occuring in the result of the elaboration will be a subset of the names occuring in the input and output name sets, i.e. the state will correctly record the set of generated names. In the type-theoretic semantics, well-formedness of the context ensures that the free names of the result are a subset of the free names of the context:

Lemma 3 (Free Names). *If* $C\,\mathrm{WF}$ *then* $C \vdash d/\mathit{se} : X$ *implies* $\mathrm{FN}(X) \subseteq \mathrm{FN}(C)$.

Proof(Sketch). The proof follows easily by rule induction. The idea is to maintain well-formedness as an invariant of the system and appeal to Lemma 2 in the functor application rule. □

5 Main Result

Having defined our systems, we are now in a position to state the main result of the paper:

Theorem 4 (Main Result). *Provided* $C\,\mathrm{WF}$ *and* $C, N\,\mathrm{rigid}$*:*

Completeness *If* $C, N \vdash d/\mathit{se} \Rightarrow S, M$ *then* $C \vdash d/\mathit{se} : \exists M.S$.

Soundness *If* $C \vdash d/\mathit{se} : X$ *then, for some* M *and* S, $C, N \vdash d/\mathit{se} \Rightarrow S, M$ *with* $X \stackrel{\alpha}{\equiv} \exists M.S$.

An operational view of the system in Figure 4 is that we have replaced the notion of *global* generativity by *local* generativity and the ability to rename bound names when necessary. The proof of completeness is easy because, if a name is globally fresh, it will certainly be locally fresh, enabling a straightfoward construction of a corresponding derivation.

Proof(Completeness). We use simultaneous induction on the elaboration rules to prove the theorems:

$$C, N \vdash d/\mathit{se} \Rightarrow S, M \supset C\,\mathrm{WF} \supset C, N\,\mathrm{rigid} \supset C \vdash d/\mathit{se} : \exists M.S$$

For lack of space, we will only consider the rule for structure declarations:

$\boxed{\text{Rule (E-4)}}$ Assume the premises

$$C, N \vdash se \Rightarrow S, P \text{ (i)} \qquad\qquad C[x : S], N \cup P \vdash d \Rightarrow S', Q \text{ (ii)}$$

and induction hypotheses:

$$C\,\mathrm{WF} \supset C, N\,\mathrm{rigid} \supset C \vdash se : \exists P.S \qquad \text{(iii)}$$
$$C[x : S]\,\mathrm{WF} \supset C[x : S], N \cup P\,\mathrm{rigid} \supset C[x : S] \vdash d : \exists Q.S' \qquad \text{(iv)}$$

Suppose $C\,\mathrm{WF}$ (v) and $C, N\,\mathrm{rigid}$ (vi). Using induction hypothesis (iii) on (v) and (vi) we obtain:

$$C \vdash se : \exists P.S \qquad \text{(vii)}$$

Property 1 of (i), together with (vi), ensures that:

$$P \cap \mathrm{FN}(C) = \emptyset \qquad \text{(viii)}$$

Clearly (v) extends to $C[x : S]\,\mathrm{WF}$ (ix).
Hence Lemma 3 on (vii) guarantees $\mathrm{FN}(\exists P.S) \subseteq \mathrm{FN}(C)$.
It follows from (vi) that $\mathrm{FN}(S) \subseteq N \cup P$ (x) and consequently $C[x : S], N \cup P\,\mathrm{rigid}$ (xi).
Applying the induction hypothesis (iv) to (ix) and (vi) yields:

$$C[x : S] \vdash d : \exists Q.S' \qquad \text{(xii)}$$

Property 1 of (ii) ensures $Q \cap (N \cup P) = \emptyset$ which, together with (x), entails:

$$Q \cap (P \cup \mathrm{FN}(S)) = \emptyset \qquad \text{(xiii)}$$

Rule (T-4) on (vii), (viii), (xiii) and (xii) derives

$$C \vdash \textbf{structure}\ x = se; d : \exists P \cup Q.[x : S]S'$$

as desired. $\square$

In the complete proof, Property 1 and Lemma 3 conspire to ensure the side conditions of Rules (T-1)—(T-8) hence appeals to the α-conversion Rules (T-9) and (T-10) are never required.

5.1 Soundness

Soundness is more difficult to prove, because the type system in Figure 4 only requires subderivations to hold for *particular* choices of locally fresh names. A name may be locally fresh without being globally fresh, foiling naive attempts to construct an elaboration from a typing derivation.

$$\boxed{C \vdash' d : X} \qquad \frac{C \vdash te \triangleright \tau \quad \delta \notin Q \quad \forall\gamma.C[t=\gamma] \vdash' d : [\gamma/\delta]\langle\exists Q.S\rangle}{C \vdash' \textbf{datatype}\ t\ =\ te; d : \exists\{\delta\} \cup Q.[t=\delta]S} \text{ (T'-3)}$$

$$\frac{\begin{array}{c} C \vdash' se : \exists P.S \quad Q \cap (P \cup \mathrm{FN}(S)) = \emptyset \\ \forall\pi.\mathrm{Dom}(\pi) = P \supset C[x\ :\ \pi\langle S\rangle] \vdash' d : \pi\langle\exists Q.S'\rangle \end{array}}{C \vdash' \textbf{structure}\ x\ =\ se; d : \exists P \cup Q.[x\ :\ S]S'} \text{ (T'-4)}$$

$$\frac{\begin{array}{ll} C \vdash Sg \triangleright \Lambda P.S & P \cap \mathrm{FN}(C) = \emptyset \\ \forall\pi.\mathrm{Dom}(\pi) = P \supset C[x\ :\ \pi\langle S\rangle] \vdash' se : \pi\langle X'\rangle & C[f\ :\ \forall P.S \to X'] \vdash' d : X \end{array}}{C \vdash' \textbf{functor}\ f\ (x : Sg)\ =\ se\ \textbf{in}\ d : X} \text{ (T'-5)}$$

$$\boxed{C \vdash' se : X} \qquad \frac{\begin{array}{c} C \vdash' se : \exists P.S' \quad P \cap (\mathrm{FN}(C(f)) \cup Q) = \emptyset \\ \forall\pi.\mathrm{Dom}(\pi) = P \supset \exists S''.S'' \to \pi\langle\exists Q.S\rangle \leq C(f) \wedge \pi\langle S'\rangle \succeq S'' \end{array}}{C \vdash' f\ se : \exists P \cup Q.S} \text{ (T'-7)}$$

Figure 5: Type system with generalised premises

To address this problem, we introduce a modified formulation of the type system with the judgement forms $C \vdash' d : \exists M.S$ and $C \vdash' se : \exists M.S$. The modified rules appear in Figure 5. Instead of requiring premises to hold for *particular* choices of fresh names, they require them to hold for *every* choice of names. This makes it easy to reconstruct an elaboration tree from a judgement in the generalised system. Note that the inference rules are no longer finitely branching, yet they remain well-founded and amenable to inductive arguments. The technique is adapted from McKinna and Pollack's formalisation of α-conversion[12].

The strategy for proving soundness is to first show that any derivation in the original type theoretic system gives rise to a corresponding derivation in the generalised system:

Lemma 5 (Soundness — Part I). *If* $C\,\mathrm{WF}$ *and* $C \vdash d/se : X$ *then* $C \vdash' d/se : X$.

One then proves that any derivation in the generalised system gives rise to a corresponding elaboration:

Lemma 6 (Soundness — Part II). *If* $C\,\mathrm{WF}$ *and* $C \vdash' d/se : X$ *then, for any* N *satisfying* $C, N\,\mathrm{rigid}$*, we can find an* M *and* S *such that* $C, N \vdash d/se \Rightarrow S, M$*, with* $X \stackrel{\alpha}{\equiv} \exists M.S$.

Proof(Lemma 5). We use simultaneous induction on the rules to prove the *stronger* theorems:

$$C \vdash d/se : X \supset C\,\mathrm{WF} \supset \forall\rho.\rho\langle C\rangle \vdash' d/se : \rho\langle X\rangle$$

Lemma 5 follows immediately by choosing ρ to be the empty (identity) renaming.

For lack of space, we will only consider the rule for structure declarations. The other cases are similar.

Rule (T-4) Assume the premises

$C \vdash se : \exists P.S_p$ (i)	$P \cap \mathrm{FN}(C) = \emptyset$ (ii)
$C[x\ :\ S_p] \vdash d : \exists Q.S_q$ (iii)	$Q \cap (P \cup \mathrm{FN}(S_p)) = \emptyset$ (iv)

and induction hypotheses:

$$C\,\mathrm{WF} \supset \forall \rho.\rho\langle C\rangle \vdash' \mathit{se} : \rho\langle \exists P.S_p\rangle \tag{v}$$
$$C[x\ :\ S_p]\,\mathrm{WF} \supset \forall \rho.\rho\langle C[x\ :\ S_p]\rangle \vdash' d : \rho\langle \exists Q.S_q\rangle \tag{vi}$$

Suppose $C\,\mathrm{WF}$ (vii) and consider an arbitrary renaming ρ. We need to show:

$$\rho\langle C\rangle \vdash' \textbf{structure}\ x\ =\ \mathit{se}; d : \rho\langle \exists P \cup Q.[x\ :\ S_p]S_q\rangle \tag{viii}$$

Choose a fresh set of names M and structure S_m such that

$$\exists P \cup Q.[x\ :\ S_p]S_q \stackrel{\alpha}{\equiv} \exists M.S_m$$

and $M \cap (\mathrm{Inv}(\rho) \cup P \cup \mathrm{FN}(\rho\langle C\rangle)) = \emptyset$. By the definition of $\stackrel{\alpha}{\equiv}$ we must have some bijective renaming σ with domain $P \cup Q$ and range M such that $S_m \equiv \sigma\langle [x\ :\ S_p]S_q\rangle$. Let $P' \stackrel{\mathrm{def}}{=} \sigma\langle P\rangle$, $Q' \stackrel{\mathrm{def}}{=} \sigma\langle Q\rangle$, $[P'/P] \stackrel{\mathrm{def}}{=} \sigma \downarrow P$, $[Q'/Q] \stackrel{\mathrm{def}}{=} \sigma \downarrow Q$, $S'_p \stackrel{\mathrm{def}}{=} \sigma\langle S_p\rangle$ and $S'_q \stackrel{\mathrm{def}}{=} \sigma\langle S_q\rangle$. Re-expressing the previous equation we have:

$$\exists P \cup Q.[x\ :\ S_p]S_q \stackrel{\alpha}{\equiv} \exists P' \cup Q'.[x\ :\ S'_p]S'_q$$

Moreover, it is easy to verify that:

$$P' \cap Q' = \emptyset\ \text{(ix)} \qquad \exists P.S_p \stackrel{\alpha}{\equiv} \exists P'.S'_p\ \text{(x)} \qquad \exists Q.S_q \stackrel{\alpha}{\equiv} [P/P']\langle \exists Q'.S'_q\rangle\ \text{(xi)}$$

By induction hypothesis (v) on (vii) and ρ we obtain $\rho\langle C\rangle \vdash' \mathit{se} : \rho\langle \exists P.S_p\rangle$ (xii). Now $\rho\langle \exists P.S_p\rangle \stackrel{\alpha}{\equiv} \rho\langle \exists P'.S'_p\rangle \stackrel{\alpha}{\equiv} \exists P'.\rho\langle S'_p\rangle$ by (x) and since $P' \cap \mathrm{Inv}(\rho) = \emptyset$, so by α-conversion (Rule (T'-10)) on (xii) we can derive:

$$\rho\langle C\rangle \vdash' \mathit{se} : \exists P'.\rho\langle S'_p\rangle \tag{xiii}$$

By Lemma 3 on (i) and (vii) we have $\mathrm{FN}(\exists P.S_p) \subseteq \mathrm{FN}(C)$ from which it is easy to deduce that $\mathrm{FN}(\rho\langle S'_p\rangle) \subseteq P' \cup \mathrm{FN}(\rho\langle C\rangle)$. Together with (ix) and our choice of M this ensures:

$$Q' \cap (P' \cup \mathrm{FN}(\rho\langle S'_p\rangle)) = \emptyset \tag{xiv}$$

It remains to show:

$$\forall \pi.\mathrm{Dom}(\pi) = P' \supset \rho\langle C\rangle[x\ :\ \pi\langle \rho\langle S'_p\rangle\rangle] \vdash' d : \pi\langle \exists Q'.\rho\langle S'_q\rangle\rangle \tag{xv}$$

Consider an arbitrary renaming π with domain P'. Let $\rho' \stackrel{\mathrm{def}}{=} \rho + (\pi \circ [P'/P])$. Clearly (vii) extends to $C[x\ :\ S_p]\,\mathrm{WF}$. Applying induction hypotheses (vi) to ρ' establishes

$$\rho'\langle C[x\ :\ S_p]\rangle \vdash' d : \rho'\langle \exists Q.S_q\rangle \tag{xvi}$$

Now $\rho'\langle C\rangle \equiv \rho + (\pi \circ [P'/P])\langle C\rangle \equiv \rho\langle C\rangle$ since $P \cap \mathrm{FN}(C) = \emptyset$ and

$$\begin{aligned}
\rho'\langle S_p\rangle &\equiv \pi\langle \rho + [P'/P]\langle S_p\rangle\rangle && \text{by considering } \alpha \in \mathrm{FN}(S_p)\\
&\equiv \pi\langle \rho\langle [P'/P]\langle S_p\rangle\rangle\rangle && \text{since } P' \cap \mathrm{Dom}(\rho) = \emptyset\\
&\equiv \pi\langle \rho\langle \sigma\langle S_p\rangle\rangle\rangle && \text{since } Q \cap \mathrm{FN}(S_p) = \emptyset\\
&\equiv \pi\langle \rho\langle S'_p\rangle\rangle && \text{(xvii)}
\end{aligned}$$

so we can re-express (xvi) as $\rho\langle C\rangle[x : \pi\langle\rho\langle S'_p\rangle\rangle] \vdash' d : \rho'\langle \exists Q.S_q\rangle$ (xviii). Observe that

$$\begin{aligned}
\rho'\langle \exists Q.S_q\rangle &\stackrel{\alpha}{\equiv} \rho + (\pi \circ [P'/P])\langle [P/P']\langle \exists Q'.S'_q\rangle\rangle && \text{by (xi)}\\
&\equiv \pi\langle \rho + [P'/P]\langle \exists Q'.S'_q\rangle\rangle && \text{by considering } \alpha \in \mathrm{FN}(\exists Q'.S'_q)\\
&\equiv \pi\langle \rho\langle \exists Q'.S'_q\rangle\rangle && \text{since } P \cap \mathrm{FN}(\exists Q'.S'_q) = \emptyset\\
&\stackrel{\alpha}{\equiv} \pi\langle \exists Q'.\rho\langle S'_q\rangle\rangle && \text{since } Q' \cap \mathrm{Inv}(\rho) = \emptyset.
\end{aligned}$$

Using α-conversion (Rule (T'-9)) on (xviii) and the last equation we can derive:

$$\rho\langle C\rangle[x : \pi\langle\rho\langle S'_p\rangle\rangle] \vdash' d : \pi\langle \exists Q'.\rho\langle S'_q\rangle\rangle \qquad \text{(xix)}$$

Since π was arbitrary we have established (xv).
Rule (T'-4) on (xiii), (xiv) and (xv) lets us derive:

$$\rho\langle C\rangle \vdash' \textbf{structure}\ x = \mathit{se}; d : \exists P' \cup Q'.[x : \rho\langle S'_p\rangle]\rho\langle S'_q\rangle \qquad \text{(xx)}$$

Finally, $\exists P' \cup Q'.[x : \rho\langle S'_p\rangle]\rho\langle S'_q\rangle \stackrel{\alpha}{\equiv} \rho\langle \exists P \cup Q.[x : S_p]S_q\rangle$ follows easily from $(P' \cup Q') \cap \mathrm{Inv}(\rho) = \emptyset$. Applying α-conversion (Rule (T'-9)) on (xx) and the last equation yields (viii) as desired. □

Before proceeding with the proof of Lemma 6 we will require the counterpart to Lemma 3 (proof omitted but easy):

Lemma 7 (Free Names). *If* C WF *and* $C \vdash' d/\mathit{se} : X$ *then* $\mathrm{FN}(X) \subseteq \mathrm{FN}(C)$.

Proof(Lemma 6). We use simultaneous induction on the rules to prove the theorems:

$$C \vdash' d/\mathit{se} : X \supset C\,\mathrm{WF} \supset \forall N.\ C, N\,\mathrm{rigid} \supset \exists M, S.\ C, N \vdash d/\mathit{se} \Rightarrow S, M \ \wedge\ X \stackrel{\alpha}{\equiv} \exists M.S$$

Again, we will only consider the rule for structure declarations. The other cases are similar.

$\boxed{\text{Rule (T'-4)}}$ Assume the premises

$$C \vdash' \mathit{se} : \exists P.S_p \text{ (i)} \qquad Q \cap (P \cup \mathrm{FN}(S_p)) = \emptyset \text{ (ii)}$$
$$\forall \pi.\mathrm{Dom}(\pi) = P \supset C[x : \pi\langle S_p\rangle] \vdash' D : \pi\langle \exists Q.S_q\rangle \text{ (iii)}$$

and induction hypotheses:

$$\text{(iv)} \quad \begin{array}{l} C\,\mathrm{WF} \supset \\ \quad \forall N. \quad C, N\,\mathrm{rigid} \supset \\ \qquad\qquad \exists P', S'_p. \quad C, N \vdash \mathit{se} \Rightarrow S'_p, P' \\ \qquad\qquad\qquad\qquad \wedge\ \exists P.S_p \overset{\alpha}{\equiv} \exists P'.S'_p \end{array}$$

$$\text{(v)} \quad \begin{array}{l} \forall \pi.\mathrm{Dom}(\pi) = P \supset \\ \quad C[x\ :\ \pi\langle S_p\rangle]\,\mathrm{WF} \supset \\ \qquad \forall N. \quad C[x\ :\ \pi\langle S_p\rangle], N\,\mathrm{rigid} \supset \\ \qquad\qquad \exists Q', S'_q. \quad C[x\ :\ \pi\langle S_p\rangle], N \vdash d \Rightarrow S'_q, Q' \\ \qquad\qquad\qquad\qquad \wedge\ \pi\langle \exists Q.S_q\rangle \overset{\alpha}{\equiv} \exists Q'.S'_q \end{array}$$

Suppose $C\,\mathrm{WF}$ (vi) and consider an arbitrary N with $C, N\,\mathrm{rigid}$ (vii). We need to show:

$$\exists M, S.\ C, N \vdash \textbf{structure}\ x\ =\ \mathit{se}; d \Rightarrow S, M\ \ \wedge\ \ \exists P \cup Q.[x\ :\ S_p]S_q \overset{\alpha}{\equiv} \exists M.S$$

From induction hypothesis (iv) we obtain a P' and S'_p satisfying:

$$C, N \vdash \mathit{se} \Rightarrow S'_p, P'\ \text{(viii)} \qquad\qquad \exists P.S_p \overset{\alpha}{\equiv} \exists P'.S'_p\ \text{(ix)}$$

Hence we must have $[P'/P]\langle S_p\rangle \equiv S'_p$ (x) for some bijective renaming $[P'/P]$. Clearly (vi) extends to $C[x\ :\ [P'/P]\langle S_p\rangle]\,\mathrm{WF}$ (xi).
Lemma 7 on (vi) and (i) establishes $\mathrm{FN}(\exists P.S_p) \subseteq \mathrm{FN}(C)$ (xii). From (ix) and (vii) it follows that $\mathrm{FN}([P'/P]\langle S_p\rangle) \subseteq N \cup P'$ and thus $C[x\ :\ [P'/P]\langle S_p\rangle], N \cup P'\,\mathrm{rigid}$. We are now in a position to apply induction hypotheses (v) to the renaming $[P'/P]$ resulting in a Q' and S'_q with:

$$C[x\ :\ S'_p], N \cup P' \vdash d \Rightarrow S'_q, Q'\ \text{(xiii)} \qquad\qquad [P'/P]\langle \exists Q.S_q\rangle \overset{\alpha}{\equiv} \exists Q'.S'_q\ \text{(xiv)}$$

Rule (E-4) on (viii) and (xiii) derives:

$$C, N \vdash \textbf{structure}\ x\ =\ \mathit{se}; d \Rightarrow [x\ :\ S'_p]S'_q, P' \cup Q' \tag{xv}$$

We still need to show $\exists P \cup Q.[x\ :\ S_p]S_q \overset{\alpha}{\equiv} \exists P' \cup Q'.[x\ :\ S'_p]S'_q$.
First, observe that $\mathrm{FN}(S_p) \subseteq \mathrm{FN}(C) \cup P$ by (xii). Applying premise (iii) to the identity renaming on P yields $C[x\ :\ S_p] \vdash' d : \exists Q.S_q$. An application of Lemma 7 yields $\mathrm{FN}(S_q) \subseteq \mathrm{FN}(C) \cup P \cup Q$ (since (vi) extends to $C[x\ :\ S_p]\,\mathrm{WF}$). Using equations (ix) and (xiv) one easily obtains $\mathrm{FN}(S'_p) \subseteq \mathrm{FN}(C) \cup P'$ and $\mathrm{FN}(S'_q) \subseteq \mathrm{FN}(C) \cup P' \cup Q'$. Moreover, Property 1 of (viii) together with (vii) ensures $Q' \cap (\mathrm{FN}(C) \cup P') = \emptyset$.
Now choose a "fresh" bijective renaming $[\hat{Q}/Q]$ with $\hat{Q} \cap (P \cup P' \cup Q \cup Q' \cup \mathrm{FN}(C)) = \emptyset$. Then it is easy to see that

$$\exists Q.S_q \overset{\alpha}{\equiv} \exists \hat{Q}.[\hat{Q}/Q]\langle S_q\rangle$$

and consequently

$$\begin{aligned}
\exists Q'.S'_q &\stackrel{\alpha}{\equiv} [P'/P]\langle \exists Q.S_q\rangle \\
&\stackrel{\alpha}{\equiv} [P'/P]\langle \exists \hat{Q}.[\hat{Q}/Q]\langle S_q\rangle\rangle \\
&\stackrel{\alpha}{\equiv} \exists \hat{Q}.[P'/P]\langle [\hat{Q}/Q]\langle S_q\rangle\rangle.
\end{aligned}$$

Hence there is a bijection $[Q'/\hat{Q}]$ such that $S'_q \equiv [Q'/\hat{Q}]\langle [P'/P]\langle [\hat{Q}/Q]\langle S_q\rangle\rangle\rangle$.
With these observations, it is easy to show:

$$\begin{aligned}
\exists P \cup Q.[x\ :\ S_p]S_q &\stackrel{\alpha}{\equiv} \exists P \cup \hat{Q}.[x\ :\ [\hat{Q}/Q]\langle S_p\rangle][\hat{Q}/Q]\langle S_q\rangle \\
&\equiv \exists P \cup \hat{Q}.[x\ :\ S_p][\hat{Q}/Q]\langle S_q\rangle \\
&\stackrel{\alpha}{\equiv} \exists P' \cup \hat{Q}.[x\ :\ [P'/P]\langle S_p\rangle][P'/P]\langle [\hat{Q}/Q]\langle S_q\rangle\rangle \\
&\equiv \exists P' \cup \hat{Q}.[x\ :\ S'_p][P'/P]\langle [\hat{Q}/Q]\langle S_q\rangle\rangle \\
&\stackrel{\alpha}{\equiv} \exists P' \cup Q'.[x\ :\ [Q'/\hat{Q}]\langle S'_p\rangle][Q'/\hat{Q}]\langle [P'/P]\langle [\hat{Q}/Q]\langle S_q\rangle\rangle\rangle \\
&\equiv \exists P' \cup Q'.[x\ :\ S'_p]S'_q
\end{aligned}$$

Choosing $M \stackrel{\text{def}}{=} P' \cup Q'$ and $S \stackrel{\text{def}}{=} [x\ :\ S'_p]S'_q$ yields the desired result. □

6 Contribution

Theorem 4 is essentially an equivalence result. However, we feel that the type theoretic presentation of the static semantics provides a better *conceptual* understanding of the type structure of Standard ML. To illustrate, we will briefly consider some extensions of the static semantics. They are readily suggested by the type system in Figure 4, but, in the author's view, are less apparent from the elaboration semantics. We make no formal claim of type security for these extensions, but conjecture them to be sound [16].

6.1 Projections from Arbitrary Structure Expressions

At the moment, projections of type and structure components from structures are restricted to the case where we are projecting from a structure path *sp*. We can lift this restriction, as long as we ensure that hypothetical types do not escape their scope. Consider adding the structure expression *se*$.x$ and replacing the type expression *sp*$.t$ by *se*$.t$. The rules are simple to state:

$$\frac{C \vdash se : \exists P.S' \quad S'(x) = S}{C \vdash se.x : \exists P.S} \qquad \frac{C \vdash se : \exists P.S \quad S(t) = \tau \quad P \cap \mathrm{FN}(\tau) = \emptyset}{C \vdash se.t \rhd \tau}$$

For example, we now allow functor applications to appear in type expressions, as long as this does not require an abstract type to escape its scope. The extension is complete for programs in the original system: a path always type-checks to an existential structure with an empty quantifier so we are free to project any of its type components without violating $P \cap \mathrm{FN}(\tau) = \emptyset$.

6.2 Applicative Functors

Although the Standard ML approach of generating fresh names for the datatypes returned by a functor *at each application* is sufficient to ensure type safety, it is certainly not necessary. One need only observe that, because of the absence of conditional module expressions, the generative types returned by a functor can at most depend on the types imported from the argument, but never on the run-time value of the argument. Roughly speaking, if we apply a functor to two arguments with equal type components but possibly differing value components, then the types created by the applications can safely be identified [8]. To capture this intuition, we first modify the rule for existential introduction, to take account of the fact that the new name δ depends on the free names $\{\alpha_1, \ldots, \alpha_n\}$ $(n \geq 0)$ of its definition τ. The dependency is expressed by parameterising δ by $\alpha_1, \ldots, \alpha_n$ in the bindings to t, in effect skolemising the existential variable [6]:

$$\frac{C \vdash te \rhd \tau \quad \{\alpha_1, \ldots, \alpha_n\} = \mathrm{FN}(\tau) \quad \delta \notin \mathrm{FN}(C) \quad \begin{array}{r} Q \cap \{\delta, \alpha_1, \ldots, \alpha_n\} = \emptyset \\ C[t = \delta(\alpha_1, \ldots, \alpha_n)] \vdash d : \exists Q.S \end{array}}{C \vdash \textbf{datatype}\ t\ =\ te; d : \exists \{\delta\} \cup Q.[t = \delta(\alpha_1, \ldots, \alpha_n)]S}$$

Typically, the type names free in τ will just be the free names introduced by the elimination of existential quantifiers. However, after type checking the body of a functor, we discharge the parametric type names of the argument signature by universally quantifying them. With our modified rule for datatypes, all occurrences of existentials will be parameterised by the functor argument types on which they depend, allowing us to lift the existential from the functor result over the entire functor signature:

$$\frac{\begin{array}{l} C \vdash Sg \rhd \Lambda P.S_p \quad P \cap \mathrm{FN}(C) = \emptyset \quad C[x\ :\ S_p] \vdash se : \exists Q.S_q \\ Q \cap (\mathrm{FN}(C) \cup P) = \emptyset \quad C[f\ :\ \forall P.S_p \to S_q] \vdash d : \exists M.S \quad M \cap Q = \emptyset \end{array}}{C \vdash \textbf{functor}\ f\ (x : Sg)\ =\ se\ \textbf{in}\ d : \exists Q \cup M.S}$$

As an example, consider the sequence of declarations:

$$\begin{array}{lll} \textbf{functor} f(x : \textbf{sig type}\ t\ \textbf{end}) & = \textbf{struct} & \textbf{datatype}\ u = x.t \\ : \forall \alpha.[t = \alpha] \to [u = \delta(\alpha)][v = \gamma] & & \textbf{datatype}\ v = \textbf{bool} \\ & \textbf{end} & \end{array}$$

$$\begin{array}{l} \textbf{in} \\ \textbf{structure}\ x =\ f\ (\textbf{struct type}\ t = \textbf{int end}) : [u = \delta(\textbf{int})][v = \gamma]; \\ \textbf{structure}\ y =\ f\ (\textbf{struct type}\ t = \textbf{bool end}) : [u = \delta(\textbf{bool})][v = \gamma]; \\ \textbf{structure}\ z =\ f\ (\textbf{struct type}\ t = \textbf{int end}) : [u = \delta(\textbf{int})][v = \gamma]; \end{array}$$

From the annotated semantic objects[2] we can see that $x.u = z.u$, $x.u \neq y.u$, $x.u \neq \textbf{int}$, $y.u \neq \textbf{bool}$ and $x.v = z.v = y.v$. In the original system, the types would all be distinct. If we were to neglect the parameterisation of δ by α, then we would have $x.u = y.u$, which is unsound assuming $\textbf{int} \neq \textbf{bool}$.

[2] δ and γ are fresh names

Since functors now return unquantified result structures instead of existential structures, we can simplify the functor signatures of Figure 1 to Φ or $\forall N.S \to S' \in \mathrm{FunSig} \stackrel{\mathrm{def}}{=} \mathrm{NameSet} \times (\mathrm{Str} \times \mathrm{Str})$. The required changes to Definition 6 and Rule (T-8) are easy and left to the reader.

6.3 Abstract Signature Constraints

Standard ML has a construct (se:Sg) which is used to curtail the visibility of components of se to those specified in the signature Sg. Consider generalising signature expressions so that they denote families of *existential* structures (L or $\Lambda N.\exists M.S \in \mathrm{Sig} \stackrel{\mathrm{def}}{=} \mathrm{NameSet} \times \mathrm{ExStr}$) and introduce a new form of type specification in signature bodies **datatype** $t; D$ with corresponding rule:

$$\frac{C[t = \alpha] \vdash D \triangleright \Lambda P.\exists Q.S \quad \alpha \notin \mathrm{FN}(C) \cup P \cup Q}{C \vdash \textbf{datatype}\ t; D \triangleright \Lambda P.\exists \{\alpha\} \cup Q.[t = \alpha]S}$$

The required modifications to the existing rules for signature expressions are easy and left to the reader. For $L \equiv \Lambda N.X'$, define $X \leq L$ iff $X \stackrel{\alpha}{\equiv} \varphi(X')$ for some substitution φ with $\mathrm{Dom}(\varphi) = N$. Similarly, for $X \equiv \exists M.S'$ define $S \leq X$ iff $S \equiv \varphi(S')$ for some substitution φ with $\mathrm{Dom}(\varphi) = M$. Consider the generalised signature matching rule:

$$\frac{C \vdash \mathit{se} : \exists P.S_p \quad C \vdash \mathit{Sg} \triangleright L \quad P \cap \mathrm{FN}(L) = \emptyset \quad S_p \succeq S \quad S \leq \exists Q.S_q \quad \exists Q.S_q \leq L}{C \vdash (\mathit{se}{:}\mathit{Sg}) : \exists P \cup Q.S_q}$$

Matching a structure against a signature can hide both components (by only requiring $S_p \succeq S$) and the identities of specified types (by the additional quantification over Q). Note that when L has the form $\Lambda N.\exists\emptyset.S$ we obtain the rule of Standard ML which merely hides the accessibility of unspecified components, yet preserves all type identities. Interpreting the ML signature $\Lambda N.S$ as the generalised signature $\Lambda\emptyset.\exists N.S$ one obtains MacQueen's notion of *abstraction* [14] (implemented in NJ/SML), which hides the identities of generic type components. We offer a mixture of the two.

Of course, we now admit richer specifications of the arguments to functors, since we can state that some of the type components are not just arbitrary but abstract. Correspondingly, functor signatures can be generalised to the form Φ or $\forall N.X \to X' \in \mathrm{FunSig} \stackrel{\mathrm{def}}{=} \mathrm{NameSet} \times (\mathrm{ExStr} \times \mathrm{ExStr})$. The natural generalisation of the functor declaration rule is to to disallow the functor from propagating (via its result) the existentially quantified types of the argument, providing a greater control over modularity: in essence, the functor is restricted to making only local use of the formal argument's datatypes. The other changes to the rules are straightforward, although we should point out that the naive combination of applicative functors and generalised signature matching is unsound. Applying a signature constraint to the body of a functor may hide the dependencies of datatypes on parameters of the functor.

7 Related Work & Conclusion

The use of existential types for data abstraction was first explored in [15], but then rejected by MacQueen[10] as an inadequate explanation of SML modules. Instead, he suggested an account based on first-order dependent types which was expanded by Harper and Mitchell[4]. Unfortunately, static checking of dependently typed terms requires the evaluation of terms, which is undesirable in general-purpose programming languages[5]. More recent proposals [3, 7], based on the restricted use of dependent types, avoid this problem but bear little direct relationship to the semantics of SML. In the author's view, the syntax of SML, allowing occurrences of structure identifiers within type expressions, is misleading: the actual semantic objects show no evidence of first-order dependencies. Our account of generativity can be compared to that of Leroy [7, 9, 8]. Where he uses the dot notation as the elimination form for existentials [2], we treat the issue of existential elimination and the dot notation separately. We do not require syntactic representations of abstract types and can therefore accommodate local declarations in the style of SML. Leroy's notion of applicative functor is similar to ours, but admits fewer equalities between abstract types. Leroy's calculus supports higher-order functors, which we have not addressed. Biswas [1] proposes an elegant higher-order extension of a semantics similar to ours. It does not handle generativity but the simplifications to functor signatures sketched in Section 6.2 combined with the generalised dot notation of Section 6.1 are compatible with his work, promising an account [16] of higher-order functors *and* abstract types without resorting to the heavy machinery of [11]. Since we use existential, not dependent types, it is also possible to consider relaxing the stratification between Core and Modules.

To summarise, we have provided an explanation of Standard ML type generativity in terms of existential quantification. Our account is novel in that it does not use first-order dependent types. We suggest that SML modules can be understood as a combination of polymorphism (functors), existential types (generativity), and a subtyping relation on structures (enrichment). We sketched proper extensions to the semantics based on these observations.

References

[1] Sandip K. Biswas. Higher-order functors with transparent signatures. In *Proc. 22nd Symp. Principles of Prog. Lang.*, pages 154–163. ACM Press, 1995.

[2] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *IFIP TC2 working conference on programming concepts and methods*, 1990.

[3] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM Symp. Principles of Prog. Lang.*, 1994.

[4] Robert Harper and John C Mitchell. On the type structure of Standard ML. In *ACM Trans. Prog. Lang. Syst.*, volume 15(2), pages 211–252, 1993.

[5] Robert Harper, John C Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. Technical Report ECS-LFCS-90-112, Department of Computer Science, University of Edinburgh, April 1990.

[6] Stefan Kahrs. First-class polymorphism for ML. In *Europ. Symp. on Programming Languages and Systems*, 1994.

[7] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st Symp. Principles of Prog. Lang.*, pages 109–122. ACM press, 1994.

[8] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd Symp. Principles of Prog. Lang.*, pages 142–153. ACM Press, 1995.

[9] Xavier Leroy. A syntactic theory of type generativity and sharing. To appear in *Journal of Functional Programming.*, 1995.

[10] David MacQueen. Using dependent types to express modular structure. In *13th ACM Symp. on Principles of Prog. Lang.*, 1986.

[11] David MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald Sannella, editor, *Programming Languages and Systems - ESOP '94*, volume 788 of *LNCS*. Springer Verlag, 1994.

[12] James McKinna and Robert Pollack. Pure Type Sytems formalized. In *Proc. Int'l Conf. on Typed Lambda Calculi and Applications, Utrecht*, pages 289–305, 1993.

[13] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[14] Robing Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[15] John C Mitchell and Gordon D Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[16] Claudio V. Russo. *Types for program modules.* PhD thesis, Uni. of Edinburgh, *forthcoming* in 1996.