First-Class Structures for Standard ML

Claudio V. Russo*

Cambridge University Computer Laboratory, Cambridge CB2 3QG, UK
cvr21@cl.cam.ac.uk

Abstract. Standard ML is a statically typed programming language that is suited for the construction of both small and large programs. "Programming in the small" is captured by Standard ML's *Core* language. "Programming in the large" is captured by Standard ML's *Modules* language that provides constructs for organising related Core language definitions into self-contained modules with descriptive interfaces. While the Core is used to express details of algorithms and data structures, Modules is used to express the overall *architecture* of a software system. The Modules and Core languages are *stratified* in the sense that modules may not be manipulated as ordinary values of the Core. This is a limitation, since it means that the architecture of a program cannot be reconfigured according to run-time demands. We propose a novel extension of the language that allows modules to be manipulated as first-class values of the Core language. The extension greatly extends the expressive power of the language and has been shown to be compatible with both Core type inference and a separate extension to higher-order modules.

1 Introduction

Standard ML [10] is a high-level programming language that is suited for the construction of both small and large programs.

Standard ML's general-purpose *Core* language supports "programming in the small" with a rich range of types and computational constructs that includes recursive types and functions, control constructs, exceptions and references.

Standard ML's special-purpose *Modules* language supports "programming in the large". Constructed on top of the Core, the Modules language allows definitions of identifiers denoting Core language types and terms to be packaged together into possibly nested *structures*, whose components are accessed by the dot notation. Structures are *transparent*: by default, the *realisation* (i.e. implementation) of a type component within a structure is evident outside the structure. *Signatures* are used to specify the types of structures, by specifying their individual components. A type component may be specified *opaquely*, permitting a variety of realisations, or *transparently*, by equating it with a particular Core type. A structure *matches* a signature if it provides an implementation for

* This research was completed at the LFCS, Division of Informatics, University of Edinburgh under EPSRC grant GR/K63795. Thanks to Don Sannella, Healfdene Goguen and the anonymous referees.

G. Smolka (Ed.): ESOP/ETAPS 2000, LNCS 1782, pp. 336–350, 2000.

all of the specified components, and, thanks to *subtyping*, possibly more. A signature may be used to *opaquely constrain* a matching structure. This existentially quantifies over the actual realisation of type components that have opaque specifications in the signature, effectively hiding their implementation. A *functor* definition defines a polymorphic function mapping structures to structures. A functor may be *applied* to any structure that realises a subtype of the formal argument's type, resulting in a concrete implementation of the functor body.

Despite the flexibility of the Modules type system, the notion of computation at the Modules level is actually very weak, permitting only functor application, to model the linking of structures, and projection, to provide access to a structure's components. Moreover, the stratification between Core and Modules means that the stronger computational mechanisms of the Core cannot be exploited in the construction of structures. This severe limitation means that the architecture of a program cannot be reconfigured according to run-time demands. For instance, we cannot dynamically choose between the various back-ends of a cross compiler, if those back-ends are implemented as separate structures.

In this paper, we relax the Core/Modules stratification, allowing structures to be manipulated as *first-class* citizens of the Core language. Our extension allows structures to be passed as arguments to Core functions, returned as results of Core computations, stored in Core data structures and so on.

For presentation purposes, we formulate our extension for a representative toy language called Mini-SML. The static semantics of Mini-SML is based directly on that of Standard ML. Mini-SML includes the essential features of Standard ML Modules but, for brevity, only has a simple Core language of explicitly typed, monomorphic functions ([16] treats a Standard ML-like Core). Section 2 introduces the syntax of Mini-SML. Section 3 gives a motivating example to illustrate the limitations of the Core/Modules stratification. Section 4 reviews the static semantics of Mini-SML. Section 5 defines our extension to first-class structures. Section 6 revisits the motivating example to show the utility of our extension. Section 7 presents a different example to demonstrate that Mini-SML becomes more expressive with our extension. Section 8 discusses our contribution.

2 The Syntax of Mini-SML

The *type* and *term* syntax of Mini-SML is defined by the grammar in Figures 1 and 2, where $t \in$ TypId, $x \in$ ValId, $X \in$ StrId, $F \in$ FunId and $T \in$ SigId range over disjoint sets of type, value, structure, functor and signature identifiers.

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

Core Types	u ::=	t	type identifier
		$\mid$ u $\rightarrow$ u$'$ $\mid$ **int**	function space, integers
		$\mid$ sp.t	type projection
Signature Bodies	B ::=	**type** t = u; B	transparent type specification
		$\mid$ **type** t; B	opaque type specification
		$\mid$ **val** x : u; B	value specification
		$\mid$ **structure** X : S; B	structure specification
		$\mid$ ϵ_B	empty body
Signature Expressions	S ::=	**sig** B **end**	encapsulated body
		$\mid$ T	signature identifier

Fig. 1. Type Syntax of Mini-SML

Core Expressions	e ::=	x	value identifier
		$\mid$ λx : u.e $\mid$ e e$'$	function, application
		$\mid$ i $\mid$ **ifzero** e **then** e$'$**else** e$''$	integer, zero test
		$\mid$ **fix** e	fixpoint of e (recursion)
		$\mid$ sp.x	value projection
Structure Paths	sp ::=	X	structure identifier
		$\mid$ sp.X	structure projection
Structure Bodies	b ::=	**type** t = u; b	type definition
		$\mid$ **val** x = e; b	value definition
		$\mid$ **structure** X = s;b	structure definition
		$\mid$ **functor** F (X : S) = s; b	functor definition
		$\mid$ **signature** T = S; b	signature definition
		$\mid$ ϵ_b	empty body
Structure Expressions	s ::=	sp	structure path
		$\mid$ **struct** b **end**	structure body
		$\mid$ F(s)	functor application
		$\mid$ s :> S	opaque constraint

Fig. 2. Term Syntax of Mini-SML

are dependent in that subsequent specifications may refer to previous ones. A *signature expression* S encapsulates a body, or is a reference to a bound signature identifier. A structure *matches* a signature expression if it provides an implementation for all of the specified components, and possibly more.

Core expressions e describe a simple functional language extended with the projection of a value identifier from a structure path. A *structure path* sp is a reference to a bound structure identifier, or the projection of one of its substructures. A *structure body* b is a dependent sequence of definitions: subsequent definitions may refer to previous ones. A type definition abbreviates a type. Value and structure definitions bind term identifiers to the values of expressions. A functor definition introduces a named function on structures: X is the functor's formal argument, S specifies the argument's type, and s is the functor's body that may refer to X. The functor may be applied to any argument that matches S. A signature definition abbreviates a signature. A *structure expression* s evaluates to a structure. It may be a path or an encapsulated structure body, whose

```
signature Stream = sig type nat = int; type state;
                       val start: state;
                       val next: state → state;
                       val value: state → nat
                   end;
structure TwoOnwards = struct type nat = int; type state = nat;
                              val start = 2;
                              val next = λs:state.succ s;
                              val value = λs:state.s
                       end;
signature State = Stream;
structure Start = TwoOnwards:>State;
functor Next (S:State) =
       struct type nat = S.nat; type state = S.state;
              val filter = fix λfilter:state→state.
                      λs:state. ifzero mod (S.value s) (S.value S.start)
                                then filter (S.next s) else s;
              val start = filter S.start;
              val next = λs:state.filter (S.next s);
              val value = S.value
       end;
functor Value (S:State) = struct val value = S.value (S.start) end
```

Fig. 3. Using structures to implement streams and a stratified, but useless, Sieve.

type, value and structure definitions (but not functor or signature definitions) become the components of the structure. The application of a functor evaluates its body with respect to the value of the actual argument. An opaque constraint restricts the visibility of the structure's components to those specified in the signature, which the structure must match, and hides the actual realisations of type components with opaque specifications, introducing new abstract types.

By supporting local functor and signature definitions, structure bodies can play the role of Standard ML's separate top-level syntax. [18] formalises recursive datatypes, local structure definitions and transparent signature constraints.

3 Motivating Example: The Sieve of Eratosthenes

We can illustrate the limitations of the Core/Modules stratification of Mini-SML (and Standard ML) by attempting to implement the Sieve of Eratosthenes using Modules level structures as the fundamental "data structure". It is a moot point that the Sieve can be coded directly in the Core: our aim is to highlight the shortcomings of second-class modules. The example is adapted from [12].

The Sieve is a well-known algorithm for calculating the infinite list, or *stream*, of prime (natural) numbers. We can represent such a stream as a "process", defined by a specific representation *nat* of the set of natural numbers, an unspecified set *state* of internal states, a designated initial or *start* state, a transition function taking us from one state to the *next* state, and a function *value* returning the

natural number associated with each state. Reading the values off the process's sequence of states yields the stream.

Given a stream s, let $sift(s)$ be the substream of s consisting of those values not divisible by the initial value of s. Viewed as a process, the states of $sift(s)$ are just the states of s, filtered by the removal of any states whose values are divisible by the value of s's start state. The stream of primes is obtained by taking the initial value of each stream in the sequence of streams:

$$\begin{array}{ll} \textit{twoonwards} & = \mathbf{2}, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, \ldots \\ \textit{sift}(\textit{twoonwards}) & = \quad \mathbf{3}, \quad 5, \quad 7, \quad 9, \quad 11, \quad \ldots \\ \textit{sift}(\textit{sift}(\textit{twoonwards})) & = \qquad \mathbf{5}, \quad 7, \qquad 11, \quad \ldots \\ & \vdots \end{array}$$

The Sieve of Eratosthenes represents this construction as the following process. The *states* of the Sieve are streams. The Sieve's *start* state is the stream *twoonwards*. The *next* state of the Sieve is obtained by *sift*ing the current state. The *value* of each state of the Sieve is the first value of that state viewed as a stream. Observe that our description of the Sieve also describes a stream.

Consider the code in Fig. 3. Given our description of streams as processes, it seems natural to use structures matching the signature `Stream` to implement streams: e.g. the structure `TwoOnwards` implements the stream *twoonwards*. The remaining code constructs an implementation of the Sieve. The states of the Sieve are structures matching the signature `State` (i.e. `Stream`). The `Start` state of the Sieve is the structure `TwoOnwards`. The functor `Next` takes a structure `S` matching `State` and returns a sifted structure that also matches `State`. The functor `Value` returns the value of a state of the Sieve, by returning the initial value of the state viewed as a stream.

Now we can indeed calculate the value of the n^{th} prime (counting from 0):

```
structure NthValue = Value(Next(···Next(Start)···));
val nthprime = NthValue.value
```

by chaining n applications (underlined above) of the functor `Next` to `Start` and then extracting the resulting value. The problem is that we can only do this for a *fixed* n: because of the stratification of Core and Modules, it is impossible to implement the mathematical function that returns the nth state of the Sieve for an *arbitrary* n. It cannot be implemented as a Core function, even though the Core supports iteration, because the states of the Sieve are structures that do not belong to the Core language. It cannot be implemented as a Modules functor, because the computation on structures is limited to functor application and projection, which is too weak to express iteration. This means that our implementation of the Sieve is useless.

Notice also that, in this implementation, the components of the Sieve do not describe a stream in the sense of the signature `Stream`: the states of the Sieve are structures, not values of the Core and the state transition function is a functor, not a Core function. Our implementation fails to capture the impredicative description of the Sieve as a stream constructed from streams.

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

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

Fig. 4. Semantic Objects of Mini-SML

Once we allow structures as first-class citizens of the Core language, these problems disappear.

4 Review: The Static Semantics of Mini-SML

Before we can propose our extension, we need to present the static semantics, or typing judgements, of Mini-SML. Following Standard ML [10], the static semantics of Mini-SML distinguishes syntactic types of the language from their semantic counterparts, called *semantic objects*. Semantic objects play the role of types in the semantics. Figure 4 defines the semantic objects of Mini-SML. We let $\mathcal{O}$ range over all semantic objects.

Type variables $\alpha \in \mathit{Var}$ are just variables ranging over *semantic types* $u \in \mathit{Type}$. The latter are the semantic counterparts of syntactic Core types, and are used to record the denotations of type identifiers and the types of value identifiers. The symbols Λ, $\exists$ and $\forall$ are used to bind finite sets of type variables.

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

Semantic structures $\mathcal{S} \in \mathit{Str}$ are used as the types of structure identifiers and paths. A semantic structure maps type components to the types they denote,

and value and structure components to the types they inhabit. For clarity, we define the extension functions $\mathrm{t} \triangleright u, \mathcal{S} \stackrel{\text{def}}{=} \{\mathrm{t} \mapsto u\} + \mathcal{S}$, $\mathrm{x} : u, \mathcal{S} \stackrel{\text{def}}{=} \{\mathrm{x} \mapsto u\} + \mathcal{S}$, and $\mathrm{X} : \mathcal{S}, \mathcal{S}' \stackrel{\text{def}}{=} \{\mathrm{X} \mapsto \mathcal{S}\} + \mathcal{S}'$, and let $\epsilon_{\mathcal{S}}$ denote the empty structure $\emptyset$.

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

The *existential structure* $\exists P.\mathcal{S}$, on the other hand, is a quantified type: variables in P are existentially quantified in $\mathcal{S}$ and thus abstract. Existential structures describe the types of structure bodies and expression. Existentially quantified type variables are explicitly introduced by opaque constraints $\mathrm{s} :> \mathrm{S}$, and implicitly eliminated at various points in the static semantics.

A *semantic functor* $\forall P.\mathcal{S} \to \mathcal{X}$ describes the type of a functor identifier: the universally quantified variables in P are bound simultaneously in the functor's domain, $\mathcal{S}$, and its range, $\mathcal{X}$. These variables capture the type components of the domain on which the functor behaves polymorphically; their possible occurrence in the range caters for the propagation of type identities from the functor's actual argument: functors are polymorphic functions on structures.

A *context* $\mathcal{C}$ maps type and signature identifiers to the types and signatures they denote, and maps value, structure and functor identifiers to the types they inhabit. For clarity, we define the extension functions $\mathcal{C}, \mathrm{t} \triangleright u \stackrel{\text{def}}{=} \mathcal{C} + \{\mathrm{t} \mapsto u\}$, $\mathcal{C}, \mathrm{T} \triangleright \mathcal{L} \stackrel{\text{def}}{=} \mathcal{C} + \{\mathrm{T} \mapsto \mathcal{L}\}$, $\mathcal{C}, \mathrm{x} : u \stackrel{\text{def}}{=} \mathcal{C} + \{\mathrm{x} \mapsto u\}$, $\mathcal{C}, \mathrm{X} : \mathcal{S} \stackrel{\text{def}}{=} \mathcal{C} + \{\mathrm{x} \mapsto \mathcal{S}\}$, and $\mathcal{C}, \mathrm{F} : \mathcal{F} \stackrel{\text{def}}{=} \mathcal{C} + \{\mathrm{F} \mapsto \mathcal{F}\}$.

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

The operation of applying a realisation to a type (substitution) is extended to all semantic objects in the usual, capture-avoiding way.

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

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

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

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

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

The static semantics of Mini-SML is defined by the denotation judgements in Fig. 5 that relate type phrases to their denotations, and the classification judgements in Fig. 6 that relate term phrases to their semantic types. A complete presentation and detailed explanation of these rules may be found in [18, 16, 17].

$$\boxed{\mathcal{C} \vdash \mathrm{u} \triangleright u} \quad \frac{\mathrm{t} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{t} \triangleright \mathcal{C}(\mathrm{t})} \quad \frac{\mathcal{C} \vdash \mathrm{u} \triangleright u \quad \mathcal{C} \vdash \mathrm{u}' \triangleright u'}{\mathcal{C} \vdash \mathrm{u} \to \mathrm{u}' \triangleright u \to u'} \quad \frac{}{\mathcal{C} \vdash \mathbf{int} \triangleright \mathrm{int}} \quad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{t} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.t} \triangleright \mathcal{S}(\mathrm{t})}$$

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

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

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

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

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

$$\boxed{\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}} \quad \frac{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}}{\mathcal{C} \vdash \mathbf{sig}\ \mathrm{B}\ \mathbf{end} \triangleright \mathcal{L}} \qquad \frac{\mathrm{T} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{T} \triangleright \mathcal{C}(\mathrm{T})}$$

Fig. 5. Denotation Judgements

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

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

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

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

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

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

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L} \quad \mathcal{C}, \mathrm{T} \triangleright \mathcal{L} \vdash \mathrm{b} : \mathcal{X}}{\mathcal{C} \vdash (\mathbf{signature}\ \mathrm{T}\ =\ \mathrm{S};\ \mathrm{b}) : \mathcal{X}} \qquad \frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \exists\emptyset.\epsilon_{\mathcal{S}}}$$

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

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

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

Fig. 6. Classification Judgements (some rules for Core expressions omitted)

5 Package Types

The motivation for introducing first-class structures is to extend the range of computations on structures. One way to do this is to extend structure expressions, and thus computation at the Modules level, with the general-purpose computational constructs usually associated with the Core. Instead of complicating the Modules language in this way, we propose to maintain the distinction between Core and Modules, but relax the stratification. Our proposal is to extend the Core language with a family of Core types, called *package types*, corresponding to first-class structures. A package type is introduced by encapsulating, or *packing*, a structure as a Core value. A package type is eliminated by breaking an encapsulation, *opening* a Core value as a structure in the scope of another Core expression. Because package types are ordinary Core types, packages are first-class citizens of the Core. The introduction and elimination phrases allow computation to alternate between computation at the level of Modules and computation at the level of the Core, without having to identify the notions of computation.

Our extension requires just three new syntactic constructs, all of which are additions to the Core language:

Core Types	u ::=...	| <S>	package type
Core Expressions	e ::=...	| **pack** s **as** S	package introduction
		| **open** e **as** X : S **in** e′	package elimination

The syntactic Core type <S>, which we call a *package type*, denotes the type of a Core expression that evaluates to an encapsulated structure value. The actual type of this structure value must match the signature S: i.e. if S denotes $\Lambda P.\mathcal{S}$, then the type of the encapsulated structure must be a subtype of $\varphi(\mathcal{S})$, for φ a realisation with $\mathcal{D}(\varphi) = P$. Two package types <S> and <S′> will be equivalent if their denotations (not just their syntactic forms) are equivalent.

The Core expression **pack** s **as** S introduces a value of package type <S>. Assuming a call-by-value dynamic semantics, the phrase is evaluated by evaluating the structure expression s and encapsulating the resulting structure value as a Core value. The static semantics needs to ensure that the type of the structure expression matches the signature S. Note that two expressions **pack** s **as** S and **pack** s′ **as** S may have the same package type <S> even when the actual types of s and s′ differ (i.e. the types both match the signature, but in different ways).

The Core expression **open** e **as** X : S **in** e′ eliminates a value of package type <S>. Assuming a call-by-value dynamic semantics, the expression e is evaluated to an encapsulated structure value, this value is bound to the structure identifier X, and the value of the entire phrase is obtained by evaluating the client expression e′ in the extended environment. The static semantics needs to ensure that e has the package type <S> and that the type of e′ does not vary with the actual type of the encapsulated structure X.

The semantic Core types of Mini-SML must be extended with the semantic counterpart of syntactic package types. In Mini-SML, the type of a structure expression is an existential structure $\mathcal{X}$ determined by the judgement form $\mathcal{C} \vdash$

$\mathrm{s} : \mathcal{X}$. Similarly, the denotation of a package type, which describes the type of an encapsulated structure value, is just an encapsulated existential structure:

$$u \in \mathit{Type} ::= \ldots \mid <\mathcal{X}> \qquad \text{semantic package type}$$

We identify semantic package types that are equivalent up to matching:

Definition 4 (Equivalence of Semantic Package Types) *Two semantic package types* $<\exists P.\mathcal{S}>$ *and* $<\exists P'.\mathcal{S}'>$ *are* equivalent *if, and only if,*

- $P' \cap \mathcal{V}(\exists P.\mathcal{S}) = \emptyset$ *and* $\mathcal{S}' \succeq \varphi(\mathcal{S})$ *for some realisation* φ *with* $\mathcal{D}(\varphi) = P$*;*
- $P \cap \mathcal{V}(\exists P'.\mathcal{S}') = \emptyset$ *and* $\mathcal{S} \succeq \varphi'(\mathcal{S}')$ *for some realisation* φ' *with* $\mathcal{D}(\varphi') = P'$*.*

The following rules extend the Core judgements $\mathcal{C} \vdash \mathrm{u} \triangleright u$ and $\mathcal{C} \vdash \mathrm{e} : u$:

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S}}{\mathcal{C} \vdash <\mathrm{S}> \triangleright <\exists P.\mathcal{S}>} \tag{1}$$

Rule 1 relates a syntactic package type to its denotation as a semantic package type. The parameters of the semantic signature $\Lambda P.\mathcal{S}$ stem from opaque type specifications in S and determine the quantifier of the package type $<\exists P.\mathcal{S}>$.

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda Q.\mathcal{S}' \quad P \cap \mathcal{V}(\Lambda Q.\mathcal{S}') = \emptyset \quad \mathcal{S} \succeq \varphi(\mathcal{S}') \quad \mathcal{D}(\varphi) = Q}{\mathcal{C} \vdash (\mathbf{pack}\ \mathrm{s}\ \mathbf{as}\ \mathrm{S}) : <\exists Q.\mathcal{S}'>} \tag{2}$$

Rule 2 is the introduction rule for package types. Provided s has existential type $\exists P.\mathcal{S}$ and S denotes the semantic signature $\Lambda Q.\mathcal{S}'$, the existential quantification over P is eliminated in order to verify that $\mathcal{S}$ matches the signature. The side condition $P \cap \mathcal{V}(\Lambda Q.\mathcal{S}') = \emptyset$ prevents the capture of free variables in the signature by the bound variables in P and ensures that these variables are treated as hypothetical types. The semantic signature $\Lambda Q.\mathcal{S}'$ describes a family of semantic structures and the requirement is that the type $\mathcal{S}$ of the structure expression enriches, i.e. is a subtype of, some member $\varphi(\mathcal{S}')$ of this family. In the resulting package type $<\exists Q.\mathcal{S}'>$, the existential quantification over Q hides the actual realisation, rendering type components specified opaquely in S abstract. Because the rule merely requires that $\mathcal{S}$ is a subtype of $\varphi(\mathcal{S}')$, the package **pack** s **as** S may have fewer components than the actual structure s.

$$\frac{\mathcal{C} \vdash \mathrm{e} : <\exists P.\mathcal{S}> \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \quad P \cap \mathcal{V}(\mathcal{C}) = \emptyset \quad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{e}' : u \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash (\mathbf{open}\ \mathrm{e}\ \mathbf{as}\ \mathrm{X} : \mathrm{S}\ \mathbf{in}\ \mathrm{e}') : u} \tag{3}$$

Rule 3 is the elimination rule for package types. Provided e has package type $<\exists P.\mathcal{S}>$, where this type is determined by the denotation of the explicit syntactic signature S, the client e' of the package is classified in the extended context $\mathcal{C}, \mathrm{X} : \mathcal{S}$. The side-condition $P \cap \mathcal{V}(\mathcal{C}) = \emptyset$ prevents the capture of free variables in $\mathcal{C}$ by the bound variables in P and ensures that these variables are treated as hypothetical types for the classification of e'. By requiring that e' is

```
structure Sieve =
 struct type nat = TwoOnwards.nat; type state = <Stream>;
        val start = pack TwoOnwards as Stream;
        val next = λs:state.open s as S:Stream in pack Next(S) as Stream;
        val value = λs:state.open s as S:Stream in S.value S.start
 end;
val nthstate = fix λnthstate:int->Sieve.state.
                   λn:int.ifzero n then Sieve.start
                          else Sieve.next (nthstate (pred n));
val nthprime = λn:int.Sieve.value (nthstate n);
```

Fig. 7. The Sieve implemented using package types.

polymorphic in P, the actual realisation of these hypothetical types is allowed to vary with the value of e. Moreover, because $\mathcal{S}$ is a generic structure matching the signature S, the rule ensures that e′ does not access any components of X that are not specified in S: thus the existence of any unspecified components is allowed to vary with the actual value of e. Finally, the side condition $P \cap \mathcal{V}(u) = \emptyset$ prevents any variation in the actual realisation of P from affecting the type of the phrase.

Observe that the explicit signature S in the term **open** e **as** X : S **in** e′ uniquely determines the Core type of the expression e. This is significant for an implicitly typed language like Standard ML's Core: the explicit signature ensures that the type inference problem for that Core remains tractable and has principal solutions. Intuitively, the type inference algorithm [16] never has to *guess* the type of an expression that is used as a package. The explicit signature in the term **pack** s **as** S ensures that the package type of the expression corresponds to a well-formed signature (this may not be the case for the actual type of s): testing the equivalence of such well-formed package types (even modulo unification) can be performed by two appeals to a signature matching algorithm [16].

Rules 2 and 3 are closely related to the standard rules for second-order existential types in Type Theory [12]. The main difference, aside from manipulating n-ary, not just unary, quantifiers is that these rules also mediate between the universe of Module types and the universe of Core types. [16, 18] sketch proofs of type soundness for package types; [18] discusses implementation issues.

6 The Sieve Revisited

The addition of package types allows us to define the structure `Sieve` implementing the Sieve of Eratosthenes (Fig. 7). The Core type `Sieve.state` is the type of packaged streams `<Stream>`. The Core value `Sieve.start` is the packaged stream `TwoOnwards`. The Core function `Sieve.next` returns the next state of `Sieve` by opening the supplied state, sifting the encapsulated stream, and packaging the resulting stream as a Core value. The Core function `Sieve.value` returns the first value of its encapsulated stream argument.

It is easy to verify that `Sieve` has type:

$$\exists\emptyset.(\texttt{nat}\triangleright \text{int}, \texttt{state}\triangleright u, \texttt{start}\colon u, \texttt{next}\colon u \to u, \texttt{value}\colon u \to \text{int}),$$

where $u \equiv <\exists\{\alpha\}.(\texttt{nat}\triangleright \text{int}, \texttt{state}\triangleright \alpha, \texttt{start}\colon \alpha, \texttt{next}\colon \alpha \to \alpha, \texttt{value}\colon \alpha \to \text{int})>$ is the type of packed streams.

`Sieve` elegantly captures the impredicative description of the Sieve as a *stream* constructed from streams: its type also matches `Stream`, since

$$(\texttt{nat}\triangleright \text{int}, \texttt{state}\triangleright u, \texttt{start}\colon u, \texttt{next}\colon u \to u, \texttt{value}\colon u \to \text{int}) \succeq \{\alpha \mapsto u\}\,(\texttt{nat}\triangleright \text{int}, \texttt{state}\triangleright \alpha, \texttt{start}\colon \alpha, \texttt{next}\colon \alpha \to \alpha, \texttt{value}\colon \alpha \to \text{int}).$$

`Sieve` is a useful implementation because it allows us to define the functions `nthstate` and `nthprime` of Fig. 7. Since the states of `Sieve` are just ordinary Core values, which happen to have package types, the function `nthstate` n can use recursion on n to construct the n^{th} state of `Sieve`. In turn, this permits the function `nthprime` n to calculate the n^{th} prime, for an *arbitrary* n. Recall that, in the absence of package types, these functions could not be defined using the implementation of the Sieve we gave in Section 3.

7 Another Example: Dynamically-Sized Arrays

Package types permit the actual realisation of an abstract type to depend on the result of a Core computation. For this reason, package types strictly extend the class of abstract types that can be defined in vanilla Mini-SML.

A familiar example of such a type is the type of *dynamically* allocated arrays of size n, where n is a value that is computed at run-time. For simplicity, we implement *functional* arrays of size 2^n, for arbitrary $n \geq 0$ (Fig. 8).

The signature `Array` specifies structures implementing integer arrays with the following interpretation. For a fixed n, the type `array` represents arrays containing 2^n entries of type `entry` (equivalent to `int`). The function `init` e creates an array that has its entries initialised to the value of e. The function `sub` a i returns the value of the (i mod 2^n)-th entry of the array a. The function `update` a i e returns an array that is equivalent to the array a, except for the (i mod 2^n)-th entry that is updated with the value of e. Interpreting each index i modulo 2^n allows us to omit array bound checks.

The structure `ArrayZero` implements arrays of size $2^0 = 1$. An array is represented by its sole entry with trivial `init`, `sub` and `update` functions.

The functor `ArraySucc` maps a structure `A`, implementing arrays of size 2^n, to a structure implementing arrays of size 2^{n+1}. The functor represents an array of size 2^{n+1} as a pair of arrays of size 2^n. Entries with even (odd) indices are stored in the first (second) component of the pair. The function `init` e returns a pair of initialised arrays of size 2^n. The function `sub` a i (`update` a i e) uses the parity of i to determine which subarray to subscript (update).

The Core function `mkArray` n uses recursion on n to construct a package implementing arrays of size 2^n. Notice that the actual realisation of the abstract type `array` returned by `mkArray` n is a balanced, nested cross product of depth n: the shape of this type depends on the run-time value of n. Interestingly, this is an example of "data-structural bootstrapping" [14] yet does not use non-regular recursive types or polymorphic recursion: it does not even use recursive types!

```
signature Array = sig type entry = int; type array; (*array is opaque*)
                      val init: entry → array;
                      val sub: array → int → entry;
                      val update: array → int → entry → array
                  end;
structure ArrayZero = struct type entry = int; type array = entry;
                             val init = λe:entry.e;
                             val sub = λa:array.λi:int.a;
                             val update = λa:array.λi:int.λe:entry.e
                      end;
functor ArraySucc (A:Array) =
   struct type entry = A.entry; type array = A.array * A.array;
          val init = λe:entry. (A.init e, A.init e)
          val sub = λa:array.λi:int.
               ifzero mod i 2 then A.sub (fst a) (div i 2)
                              else A.sub (snd a) (div i 2);
          val update = λa:array.λi:int.λe:entry.
               ifzero mod i 2 then (A.update (fst a) (div i 2) e, snd a)
                              else (fst a, A.update (snd a) (div i 2) e)
   end;
val mkArray = fix λmkArray:int→<Array>.
               λn:int. ifzero n then pack ArrayZero as Array
                       else open mkArray (pred n) as A:Array in
                                pack ArraySucc(A) as Array;
```

Fig. 8. `mkArray` n returns an abstract implementation of arrays of size 2^n.

8 Contribution

For presentation purposes, we restricted our attention to an explicitly typed, monomorphic Core language and a first-order Modules language. In [16], we demonstrate that the extension with package types may also be applied to a Standard ML-like Core language that supports the definition of type constructors and implicitly typed, polymorphic values. For instance, Section 7.3 of [16] generalises the example of Section 7 to an implementation of dynamically sized *polymorphic* arrays where `array` is a unary type constructor taking the type of entries as an argument and the array operations are suitably polymorphic. Moreover, this extension is formulated with respect to a higher-order Modules calculus that allows functors, not just structures, to be treated as first class citizens of the Modules language and, via package types, the Core language too. This proposal is practical: we present a well-behaved algorithm that integrates type inference for the extended Core with type checking for higher-order Modules. First-class and higher-order modules are available in Moscow ML V2.00[15].

Our approach to obtaining first-class structures is novel because it leaves the Modules language unchanged, relies on a simple extension of the Core language only and avoids introducing subtyping in the Core type system, which would otherwise pose severe difficulties for Core-ML type inference. (Although Mitchell *et al.* [11, 5] first suggested the idea of coercing a structure to a first-class

existential type they did not require explicit introduction and elimination terms: our insistence on these terms enables Core-ML type inference.) Our work refutes Harper and Mitchell's claim [2] that the existing type structure of Standard ML cannot accommodate first-class structures without sacrificing the compile-time/run-time phase distinction and decidable type checking. This is a limitation of their proposed model, which is based on first-order dependent types, but does not transfer to the simpler, second-order type theory [17] of Standard ML.

Our motivation for introducing first-class structures was to extend the range of computations on structures. One way to achieve this is to extend structure expressions directly with computational constructs usually associated with the Core. Taken to the extreme, this approach relaxes the stratification between Modules and the Core by removing the distinction between them, amalgamating both in a single language. This is the route taken by Harper and Lillibridge [1, 8]. Unfortunately, the identification of Core and Modules types renders subtyping, and thus type checking, undecidable. Leroy [6] briefly considers this approach without formalising it but observes that the resulting interaction between Core and Modules computation violates the type soundness of *applicative* functors [7]. Odersky and Läufer [13] and Jones [4] adopt a different tack and extend implicitly typed Core-ML with impredicative type quantification and higher-order type constructors that can model some, but not all, of the features of Standard ML Modules while providing first-class and higher-order modules.

Our approach is different. We maintain the distinction between Core and Modules, but relax the stratification by extending the Core language with package types. The introduction and elimination phrases for package types allow computation to alternate between computation at the level of Modules and computation at the level of the Core, without having to identify the notions of computation. This is reflected in the type system in which the Modules and Core typing relations are distinct. This is a significant advantage for implicitly typed Core languages like Core-ML. At the Modules level, the explicitly typed nature of Modules makes it possible to accommodate subtyping, functors with polymorphic arguments and true type constructors in the type checker for the typing relation. At the Core-ML level, the absence of subtyping, the restriction that ML functions may only take monomorphic arguments and that ML type variables range over types (but not type constructors) permits the use of Hindley-Milner [3, 9] type inference. In comparison, the amalgamated languages of [1, 8] support type constructors and subtyping, but at the cost of an explicitly typed Core fragment; [13, 4] support partial type inference, but do not provide subtyping on structures, type components in structures or a full treatment of type constructors, whose expansion and contraction must be mediated by explicit Core terms instead of implicit β-conversion.

Although not illustrated here, the advantage of distinguishing between Modules computation and Core computation is that they can be designed to satisfy different invariants [16]. For instance, the invariant needed to support applicative functors [7, 16], namely that the abstract types returned by a functor depend only on its type arguments and not the value of its term argument, is violated

if we extend Modules computation directly with general-purpose computational constructs. Applicative functors provide better support for programming with higher-order Modules; general-purpose constructs are vital for a useful Core. In [16], we show that maintaining the separation between Modules and Core computation accommodates both applicative functors and a general-purpose Core, without violating type soundness. Type soundness is preserved by the addition of package types, because these merely extend the computational power of the Core, not Modules (package elimination is weaker than including Core expressions in Module expressions). The languages of [1, 8] have higher-order functors, but their single notion of computation implies a trade-off between supporting either *applicative* functors or general-purpose computation. Since ruling out the latter is too restrictive, the functors of these calculi are not applicative.

References

[1] R. Harper, M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM Symp. Principles of Prog. Lang.*, 1994.
[2] R. Harper, J. C. Mitchell. On the type structure of Standard ML. In *ACM Trans. Prog. Lang. Syst.*, volume 15(2), pages 211–252, 1993.
[3] J. R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. of the American Mathematical Society*, 146:29-40, 1969.
[4] M. Jones. Using Parameterized Signatures to Express Modular Structure. In *23rd ACM Symp. Principles of Prog. Lang.*, 1996.
[5] D. Katiyar, D. Luckham, J. Mitchell. A type system for prototyping languages. In *24th ACM Symp. Principles of Prog. Lang.*, 1994.
[6] X. Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st Symp. Principles of Prog. Lang.*, pages 109–122. ACM Press, 1994.
[7] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd Symp. Principles of Prog. Lang.*, pages 142–153. ACM Press, 1995.
[8] M. Lillibridge. Translucent Sums: A Foundation for Higher-Order Module Systems. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
[9] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.
[10] R. Milner, M. Tofte, R. Harper, D. MacQueen. The Definition of Standard ML (Revised). MIT Press, 1997.
[11] J. C. Mitchell, S. Meldal, N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *18th ACM Symp. Principles of Prog. Lang.*, 1991.
[12] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.
[13] M. Odersky, K. Läufer. Putting Type Annotations To Work In *23rd ACM Symp. Principles of Prog. Lang.*, 1996.
[14] C. Okasaki. Purely Functional Data Structures. Cambridge University Press, 1998.
[15] S. Romanenko, P. Sestoft. Moscow ML. (`www.dina.kvl.dk/~sestoft/mosml`).
[16] C. V. Russo. Types For Modules. PhD Thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1998.
[17] C. V. Russo. Non-Dependent Types For Standard ML Modules. In *1999 Int'l Conf. on Principles and Practice of Declarative Programming.*
[18] C. V. Russo. First-Class Structures for Standard ML (long version). Forthcoming Technical Report, LFCS, Division of Informatics, University of Edinburgh, 2000.