COVER PAGE

CONFERENCE SUBMISSION

The 1999 International Conference on Functional Programming (ICFP)

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

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

Title: First-Class Structures for Standard ML

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.

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

Pages: This page + 1 blank page + 21 pages (appendix included).

COVER PAGE

THIS PAGE INTENTIONALLY LEFT BLANK

First-Class Structures for Standard ML (Extended Summary)

Claudio V. Russo

LFCS, Division of Informatics, University of Edinburgh,
JCMB, KB, Mayfield Road, Edinburgh EH9 3JZ
www: `http://www.dcs.ed.ac.uk/~cvr`, email: `cvr@dcs.ed.ac.uk`
(This research has been partially supported by EPSRC grant GR/K63795)

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 [5] 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". Standard ML's special-purpose *Modules* language supports "programming in the large".

To support algorithmic programming, the Core provides a rich range of types and computational contructs that includes recursive types and functions, control constructs, exceptions and references.

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* 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 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 level of Modules is actually very weak, consisting solely of functor application, to model the linking of structures, and projection, to provide access to the components of structures. 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 is a severe limitation, since it means that the architecture of a program cannot be reconfigured according to run-time demands.

In this paper, we relax the 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, not for Standard ML, but for a representative toy language called Mini-SML. The static semantics of Mini-SML is based directly on that of Standard ML.

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. The Appendix contains a sketched dynamic semantics and proof that our extension is sound.

2 The Syntax of Mini-SML

Mini-SML includes the essential features of Standard ML Modules but, for presentation reasons, is constructed on top of a simple Core language of explicitly typed, monomorphic functions. The author's thesis [7], on which this paper is based, presents similar results for a generic Core language that encompasses ones like Standard ML's (which supports the definition of parameterised types, is implicitly typed, and polymorphic). The *type* and *term* syntax of Mini-SML is defined by the grammar in Figures 1 and 2, where $\mathrm{t} \in \mathrm{TypId}$, $\mathrm{x} \in \mathrm{ValId}$, $\mathrm{X} \in \mathrm{StrId}$, $\mathrm{F} \in \mathrm{FunId}$ and $\mathrm{T} \in \mathrm{SigId}$ 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

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

Fig. 1. Type Syntax of Mini-SML

with the projection sp.t of a type component from a structure path. A *signature body* B is a sequential specification of a structure's components. A type component may be specified *transparently*, by equating it with a type, or *opaquely*, permitting a variety of *realisations*. Value and structure components are specified by their type and signature. The specifications in a body are dependent in that subsequent specifications may refer to previous ones. A *signature expression* 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 type, value and structure definitions become the components of the structure. The application of a functor evaluates its body with respect to the value of the actual argument. 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 realisations of type components with opaque specifications.

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

Core Expressions	e ::=	x	value identifier
		$\mid$ λx : u.e	function
		$\mid$ e e$'$	application
		$\mid$ i	integer constant
		$\mid$ **ifzero** e **then** e$'$**else** e$''$	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

3 Motivating Example: the Sieve of Eratosthenes

We can illustrate the limitations of Mini-SML, and thus Standard ML, by attempting to implement the Sieve of Eratosthenes. The example is adapted from Mitchell and Plotkin [6].

The Sieve is a well-known algorithm for enumerating prime numbers. Let *Primes* denote the enumeration $2, 3, 5, 7, 11 \ldots$ of all primes.

We can think of an enumeration of integers as a *stream*, or infinite list, of integers. In turn, we can represent such a stream as a "process", defined by an unspecified set of internal *states*, a designated initial or *start* state, a transition function taking us from one state to the *next* state, and a specific integer *value* associated with each state. Reading the values off the process's sequence of states yields the stream.

Given a stream s, let $\mathit{sift}(s)$ be the substream of s consisting of those values not divisible by the initial value of s. Viewed as a process, the states of $\mathit{sift}(s)$ 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.

If *twoonwards* is the stream $2, 3, 4, 5, 6, 7, 8, \ldots$, then the stream obtained by taking the initial value of each stream in the sequence of streams:

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

```
signature Stream = sig type state;
                       val start: state;
                       val next: state → state;
                       val value: state → int
                   end;

structure TwoOnwards = struct type state = int;
                              val start = 2;
                              val next = λi:int.succ i;
                              val value = λi:int.i
                       end;
```

Fig. 3. Using structures to implement streams.

yields the stream of primes *Primes*.

This is the intuition for constructing the Sieve of Eratosthenes. The Sieve is the following process that generates the enumeration *Primes*. The *states* of the Sieve are streams. The *start* state of the Sieve 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 Figure 3. Given our description of stream as processes, it seems natural to use structures matching the signature `Stream` to implement streams. For instance, the structure `TwoOnwards` implements the stream *twoonwards*.

The code in Figure 3 builds on these definitions to construct 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 integer value of a state of the Sieve, by returning the initial value of the state viewed as a stream. (We assume that the function `mod` i j returns the integral remainder of dividing i by j).

Using these definitions, we can indeed calculate the value of the n^{th} prime (counting from 0):

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

by chaining n applications 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

```
signature State = Stream;

structure Start = TwoOnwards:>State;

functor Next (S:State) =
      struct
         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. 4. A stratified, but useless implementation of the Sieve.

the Core language. It cannot be implemented as Modules functor, because the computation on structures is limited to functor application and projection, which is to weak to express iteration. This means that our implementation of the Sieve is effectively useless.

Notice also the discrepancy between our mathematical description of the Sieve and this implementation. In the mathematical description, the Sieve is itself a stream, just like the streams from which it is constructed. In our 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. Because of the stratification between Core and Modules, our implementation fails to capture the impredicative description of the Sieve as a stream constructed from streams.

Of course, the Sieve of Eratosthenes can be implemented directly in the Core using other means. The point is that we cannot productively use structures to represent the states of the Sieve: for this, we need to be able to perform non-trivial computations that return structures. Once we allow structures as first-class citizens of the Core language, that already supports general-purpose computation, the problem disappears.

4 Review: the Static Semantics of Mini-SML

Before we can propose our extension of Mini-SML with first-class modules, we need to briefly present the static semantics, or typing judgements, of Mini-SML.

$$
\begin{array}{rll}
\alpha \in \mathit{Var} & \stackrel{\text{def}}{=} \{\alpha, \beta, \delta, \gamma, \ldots\} & \text{type variables} \\
M,N,P,Q \in \mathit{VarSet} & \stackrel{\text{def}}{=} \mathrm{Fin}(\mathit{Var}) & \text{sets of type variables} \\
u \in \mathit{Type} & ::= \alpha & \text{type variable} \\
 & \mid\ u \to u' & \text{function space} \\
 & \mid\ \mathrm{int} & \text{integers} \\
\varphi \in \mathit{Real} & \stackrel{\text{def}}{=} \mathit{Var} \stackrel{\text{fin}}{\to} \mathit{Type} & \text{realisations} \\
\mathcal{S} \in \mathit{Str} & \stackrel{\text{def}}{=} \left\{ \begin{array}{l|l} \mathcal{S}_{\mathrm{t}} \cup & \mathcal{S}_{\mathrm{t}} \in \mathrm{TypId} \stackrel{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{S}_{\mathrm{x}} \cup & \mathcal{S}_{\mathrm{x}} \in \mathrm{ValId} \stackrel{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{S}_{\mathrm{X}} & \mathcal{S}_{\mathrm{X}} \in \mathrm{StrId} \stackrel{\text{fin}}{\to} \mathit{Str} \end{array} \right\} & \text{semantic structures} \\
\mathcal{L} \in \mathit{Sig} & ::= \Lambda P.\mathcal{S} & \text{semantic signatures} \\
\mathcal{X} \in \mathit{ExStr} & ::= \exists P.\mathcal{S} & \text{existential structures} \\
\mathcal{F} \in \mathit{Fun} & ::= \forall P.\mathcal{S} \to \mathcal{X} & \text{semantic functors} \\
\mathcal{C} \in \mathit{Context} & \stackrel{\text{def}}{=} \left\{ \begin{array}{l|l} \mathcal{C}_{\mathrm{t}} \cup & \mathcal{C}_{\mathrm{t}} \in \mathrm{TypId} \stackrel{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{C}_{\mathrm{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. 5. Semantic Objects of Mini-SML

Readers familiar with the Definition of Standard ML [5] may prefer to skim this section to assimilate the notation, and then skip ahead to Section 5.

Following Standard ML [5], the static semantics of Mini-SML distinguishes between the syntactic types of the language and their semantic counterparts called *semantic objects*. Semantic objects play the role of types in the static semantics. Figure 5 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.

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 $\emptyset$.

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

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

The *existential structure* $\exists P.\mathcal{S}$, on the other hand, is a quantified type: variables in P are existentially quantified in $\mathcal{S}$ and thus abstract. Existential structures describe the types of structure bodies and expression. Existentially quantified type variables are explicitly introduced by opaque constraints s :> 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. The range $\mathcal{X}$ is the type of the functor body.

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 way, taking care to avoid the capture of free variables by bound variables.

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 6. Denotation Judgements

The static semantics of Mini-SML is defined by the denotation judgements in Figure 6 that relate type phrases to their denotations, and the classification judgements in Figure 7 that relate term phrases to their semantic types. A detailed explanation of these rules may be found in [7,8]. We deviate from the presentation in the Definition [5] by classifying structure expressions using existentially quantified structures. The Definition classifies structure expressions using bare semantic structures, but employs procedural classification rules that maintain a state of generated type variables, updating this state each time a structure expression is constrained by an opaque signature, or a functor applied to an argument. These approaches are equivalent, but ours is stateless and more declarative [7,8].

We can illustrate the semantics by considering the semantic objects assigned to some of the phrases in Figures 3 and 4.

The denotation of `Stream` is the semantic signature:

$$\Lambda\{\alpha\}.(\mathtt{state} \triangleright \alpha, \mathtt{start} : \alpha, \mathtt{next} : \alpha \to \alpha, \mathtt{value} : \alpha \to \mathrm{int}),$$

where the parameter α arises from the opaque specification of the type `state`.

The type of the structure expression in the definition of `TwoOnwards` is:

$$\exists\emptyset.(\mathtt{state} \triangleright \mathrm{int}, \mathtt{start} : \mathrm{int}, \mathtt{next} : \mathrm{int} \to \mathrm{int}, \mathtt{value} : \mathrm{int} \to \mathrm{int})$$

for an empty existential quantifier.

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

$$\frac{\mathcal{C} \vdash \mathrm{e} : \mathrm{int} \quad \mathcal{C} \vdash \mathrm{e}' : u \quad \mathcal{C} \vdash \mathrm{e}'' : u}{\mathcal{C} \vdash \textbf{ifzero}\ \mathrm{e}\ \textbf{then}\ \mathrm{e}'\ \textbf{else}\ \mathrm{e}'' : u} \quad \frac{\mathcal{C} \vdash \mathrm{e} : (u \to u') \to u \to u'}{\mathcal{C} \vdash \textbf{fix}\ \mathrm{e} : u \to u'} \quad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{x} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.x} : \mathcal{S}(\mathrm{x})}$$

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

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

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

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

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

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \mathcal{L} \quad \mathcal{C}, \mathrm{T} \rhd \mathcal{L} \vdash \mathrm{b} : \mathcal{X}}{\mathcal{C} \vdash \textbf{signature}\ \mathrm{T}\ =\ \mathrm{S};\ \mathrm{b} : \mathcal{X}}$$

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

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

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

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

Fig. 7. Classification Judgements

The body of this type matches the signature `Stream` by choosing the realisation $\{\alpha \mapsto \mathrm{int}\}$, since:

$$\begin{array}{l}(\texttt{state} \rhd \mathrm{int}, \texttt{start} : \mathrm{int}, \texttt{next} : \mathrm{int} \to \mathrm{int}, \texttt{value} : \mathrm{int} \to \mathrm{int}) \succeq \\ \qquad \{\alpha \mapsto \mathrm{int}\}\,(\texttt{state} \rhd \alpha, \texttt{start} : \alpha, \texttt{next} : \alpha \to \alpha, \texttt{value} : \alpha \to \mathrm{int}).\end{array}$$

The signature `State` abbreviates the denotation of `Stream` so that the opaque constraint `TwoOnwards:>State` introduces the existential type:

$$\exists\{\beta\}.(\texttt{state} \rhd \beta, \texttt{start} : \beta, \texttt{next} : \beta \to \beta, \texttt{value} : \beta \to \mathrm{int}).$$

Binding this structure expression to the structure identifier `Start` eliminates the existential quantifier so that the type of `Start`, as recorded in the context, is:

$$(\texttt{state} \rhd \beta, \texttt{start} : \beta, \texttt{next} : \beta \to \beta, \texttt{value} : \beta \to \mathrm{int}),$$

for some hypothetical, and thus abstract, type β.

The type (i.e. semantic functor) of the sifting functor `Next` is the universally quantified type:

$$\begin{array}{l}\forall\{\gamma\}.(\mathtt{state} \triangleright \gamma, \mathtt{start} : \gamma, \mathtt{next} : \gamma \to \gamma, \mathtt{value} : \gamma \to \mathrm{int}) \to \\ \quad \exists\emptyset.(\mathtt{state} \triangleright \gamma, \mathtt{filter} : \gamma \to \gamma, \mathtt{start} : \gamma, \mathtt{next} : \gamma \to \gamma, \mathtt{value} : \gamma \to \mathrm{int})\end{array}$$

Because `Next` is polymorphic, it can be applied to `Start` by choosing the functor instance:

$$\begin{array}{l}(\mathtt{state} \triangleright \beta, \mathtt{start} : \beta, \mathtt{next} : \beta \to \beta, \mathtt{value} : \beta \to \mathrm{int}) \to \\ \quad \exists\emptyset.(\mathtt{state} \triangleright \beta, \mathtt{filter} : \beta \to \beta, \mathtt{start} : \beta, \mathtt{next} : \beta \to \beta, \mathtt{value} : \beta \to \mathrm{int})\end{array}$$

corresponding to the realisation $\{\gamma \mapsto \beta\}$.

The range of this functor instance determines the type of the application `Next(Start)`:

$$\exists\emptyset.(\mathtt{state} \triangleright \beta, \mathtt{filter} : \beta \to \beta, \mathtt{start} : \beta, \mathtt{next} : \beta \to \beta, \mathtt{value} : \beta \to \mathrm{int}).$$

By subtyping, this type, which is richer because it contains the additional component `filter`, also matches the signature `State`, since:

$$\begin{array}{l}(\mathtt{state} \triangleright \beta, \mathtt{filter} : \beta \to \beta, \mathtt{start} : \beta, \mathtt{next} : \beta \to \beta, \mathtt{value} : \beta \to \mathrm{int}) \succeq \\ \qquad \{\alpha \mapsto \beta\}\,(\mathtt{state} \triangleright \alpha, \mathtt{start} : \alpha, \mathtt{next} : \alpha \to \alpha, \mathtt{value} : \alpha \to \mathrm{int}).\end{array}$$

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 ::= . . .	*as before*
	| <S>	package type
Core Expressions	e ::= . . .	*as before*
	| **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 $\mathcal{D}(\varphi) = P$.

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 the two expressions **pack** s **as** S and **pack** s′ **as** S can have the same package type <S> even though the actual types of s and s′ may differ (in particular, these types may differ in the way they match the signature).

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. Note that the explicit signature determines the package type of e.

The semantic Core types of Mini-SML must be extended with the semantic counterpart of syntactic package types. In the semantics of 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:

$$\begin{array}{rcll} u \in \mathit{Type} & ::= & \ldots & \textit{as before} \\ & | & <\mathcal{X}> & \text{semantic package type} \end{array}$$

We identify package types that are equivalent up to a renaming of bound variables.

Finally, we extend the Core judgements $\mathcal{C} \vdash \mathrm{u} \rhd u$ and $\mathcal{C} \vdash \mathrm{e} : u$ with the following rules:

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{S}}{\mathcal{C} \vdash <\mathrm{S}> \rhd <\exists P.\mathcal{S}>} \qquad (1)$$

Rule 1 relates a syntactic package type to its denotation as a semantic package type. The parameters of the semantic signature $\Lambda P.\mathcal{S}$, which arise from opaque type specifications in S, simply determine the existentially quantified variables of the package type $<\exists P.\mathcal{S}>$.

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

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 $\mathcal{S}'$, the package **pack** s **as** S may have fewer components than the actual structure s.

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

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 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 subtype of $\mathcal{S}$, 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 becomes significant in the presence of an implicitly typed language like Standard ML's Core: the explicit signature ensures that the type inference problem for Standard ML's Core remains tractable and has principal solutions. Intuitively, the type inference algorithm never has to *guess* the type of an expression that is used as a package.

```
structure Sieve =
  struct
     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. 8. The Sieve implemented using package types.

Rules 2 and 3 are inspired by, and closely related to, the standard introduction and elimination rules for second-order existential types in Type Theory [6]. The main difference, aside from introducing and eliminating n-ary, not just unary, quantifiers is that these rules also mediate between the universe of Module types and the universe of Core types (see [7] for a detailed comparison).

Readers interested in the dynamic semantics of package types and a sketched proof of type soundness are referred to the Appendix.

6 The Sieve Revisited

The addition of package types allows us to define an elegant and useful implementation of the Sieve of Eratosthenes.

Figure 8 shows our implementation of the Sieve as the structure `Sieve`. The Core type `Sieve.state` is the type of packaged streams `<Stream>`. The Core value `Sieve.start` is the packaged stream `TwoOnwards` of all integers greater than 2. The Core function `Sieve.next` returns the next state of `Sieve` by opening a supplied state, applying the sifting functor `Next` to the encapsulated stream `S`, and packaging the resulting stream as a Core value. The Core function `Sieve.value` returns the value of a supplied state by opening the state and returning the first value of its encapsulated stream.

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

$$\exists\emptyset.(\texttt{state} \triangleright u, \texttt{start} : u, \texttt{next} : u \to u, \texttt{value} : u \to \mathrm{int}),$$

where u abbreviates the type of packed streams:

$$u \equiv <\exists\{\alpha\}.(\texttt{state} \triangleright \alpha, \texttt{start} : \alpha, \texttt{next} : \alpha \to \alpha, \texttt{value} : \alpha \to \mathrm{int})>.$$

The reason `Sieve` is an elegant implementation is that it captures the impredicative, mathematical description of the Sieve as a *stream* constructed from streams. This is because its type also matches the signature `Stream`, since:

$$(\texttt{state} \triangleright u, \texttt{start} : u, \texttt{next} : u \rightarrow u, \texttt{value} : u \rightarrow \text{int}) \succeq \\ \{\alpha \mapsto u\} (\texttt{state} \triangleright \alpha, \texttt{start} : \alpha, \texttt{next} : \alpha \rightarrow \alpha, \texttt{value} : \alpha \rightarrow \text{int})$$

using the realisation $\{\alpha \mapsto u\}$.

The reason `Sieve` is a useful implementation is that it allows us define the functions `nthstate` and `nthprime` of Figure 8. Because 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

With package types, it is possible to make the actual realisation of an abstract type depend on the result of some Core computation. In this way, package types strictly extend the class of abstract types that can be defined in Mini-SML alone.

A familiar example of a type whose representation depends on the result of some computation 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 $n \geq 0$ (see Figure 9).

The signature `Array` specifies structures implementing integer arrays and has the following interpretation. For a fixed n, the type `array` represents arrays containing 2^n entries of type `int`. The function `init` x creates an array that has its entries initialised to the value of x. 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 x 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 x. 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}. We assume that `div` i j returns the largest integral divisor of i by j and that the Core has been extended with the cross product type, written u `*` u$'$, supporting pairing `(`e`,`e$'$`)` and projections `fst` e and `snd` e whose static semantics are straightforward. 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` x returns a pair of initialised arrays of size 2^n. The functions `sub` a i and `update` a i x use the parity of the index i to determine which component of the array to subscript or update.

The Core function `mkArray` n, returns a package implementing arrays of size 2^n (provided $n \geq 0$). For $n = 0$, it simply returns the packaged structure `ArrayZero`. For $n \neq 0$, it first creates a package of arrays of size 2^{n-1} by recursion

```
signature Array =
 sig type array;
     val init: int → array;
     val sub: array → int → int;
     val update: array → int → int → array
 end;

structure ArrayZero =
  struct type array = int;
         val init = λx:int.x;
         val sub = λa:array.λi:int.a;
         val update = λa:array.λi:int.λx:int.x
  end;

functor ArraySucc (A:Array) =
   struct type array = A.array * A.array;
          val init = λx:int. (A.init x, A.init x)
          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.λx:int.
               ifzero mod i 2
               then (A.update (fst a) (div i 2) x, snd a)
               else (fst a, A.update (snd a) (div i 2) x)
   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. 9. `mkArray` n returns an abstract implementation of arrays of size 2^n.

on $n-1$, and then uses this package to implement a package of arrays of size 2^n by an application of the functor `ArraySucc`. Notice that the actual realisation of the abstract type `array` returned by `mkArray` depends on the value of n.

8 Contribution

For presentation purposes, we have restricted our attention to an explicitly typed, monomorphic Core language and a first-order Modules language. In [7], we demonstrate that the extension with package types may also be applied to a Standard ML-like Core language that supports the definition of parameterised types and implicitly typed, polymorphic values. 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 type checking algorithm that combines type inference for the extended Core with type checking for higher-order Modules.

Our approach to obtaining first-class structures is novel for two reasons. First, it relies on a simple extension of the Core language only, leaving the Modules language unchanged. Second, it contradicts Harper and Mitchell's claim [2] that the type structure of Standard ML cannot accommodate first-class structures without sacrificing the compile-time/run-time phase distinction and decidable type checking. While this property is true of their proposed model, which is based on first-order dependent types, the property does not transfer to Standard ML, because it has a purely second-order type theory [8].

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], and explored further in the subsequent work by Lillibridge [4]. Unfortunately, subtyping, and thus type checking, is undecidable in these calculi; they also lack the principal typing property.

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.

Although not illustrated here, the advantage of distinguishing between the two forms of computation is that they can be designed to satisfy different invariants. For instance, the invariant needed to support Leroy style *applicative* functors [3, 7], 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 good support for programming with higher-order Modules; general-purpose constructs are vital for a useful Core. In [7], we show that by keeping Modules computation and Core computation separate, we can accommodate both applicative functors and a general-purpose Core, without violating type soundness. The type soundness property is preserved by the addition of package types, because these merely extend the computational power of the Core, not Modules. By contrast, although the amalgamated languages proposed in [1, 4] have higher-order functors, because there is only a single notion of computation, there is a trade-off between supporting either *applicative* functors or general-purpose computation. Since ruling out the latter is too severe a restriction, the functors of these calculi are not applicative.

Acknowledgements: Many thanks to Don Sannella and Healfdene Goguen.

References

1. R. Harper, M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM Symp. Principles of Prog. Lang.*, 1994.
2. R. Harper, J. C. Mitchell. On the type structure of Standard ML. In *ACM Trans. Prog. Lang. Syst.*, volume 15(2), pages 211–252, 1993.
3. X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd Symp. Principles of Prog. Lang.*, pages 142–153. ACM Press, 1995.
4. M. Lillibridge. Translucent Sums: A Foundation for Higher-Order Module Systems. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
5. R. Milner, M. Tofte, R. Harper, D. MacQueen. The Definition of Standard ML (Revised). MIT Press, 1997.
6. 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.
7. C. V. Russo. Types For Modules. PhD Thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1998.
8. C. V. Russo. Non-Dependent Types For Standard ML Modules. Submitted to *1999 Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'99)*, (also available at `http://www.dcs.ed.ac.uk/~cvr`), Laboratory for Foundations of Computer Science, University of Edinburgh, 1999.

Appendix: Dynamic Semantics and Type Soundness

To demonstrate that the extension of Mini-SML with package types is sound, we need to define a dynamic semantics for the language and then prove that evaluating well-typed expressions does not lead to type violations. Although a thorough treatment of the dynamic semantics takes us beyond the scope of this paper, in this appendix, we give a brief sketch of how this might be done and use this sketch to prove the soundness of package elimination (Rule 3).

Suppose we adopt a call-by-value semantics. One way to define such a semantics, akin to the formulation of the dynamic semantics of Standard ML [5], is to define:

- a set of *core values* $\mathrm{v} \in \mathrm{CorVal}$ that includes encapsulated structure values $<\mathrm{V}>$, integer values and function closures (whose form we shall leave unspecified);
- a set of *structure values*:

$$\mathrm{V} \in \mathrm{StrVal} \stackrel{\mathrm{def}}{=} \{\mathcal{E}_{\mathrm{x}} \cup \mathcal{E}_{\mathrm{X}} \mid \mathcal{E}_{\mathrm{x}} \in \mathrm{ValId} \stackrel{\mathrm{fin}}{\to} \mathrm{CorVal},\ \mathcal{E}_{\mathrm{X}} \in \mathrm{StrId} \stackrel{\mathrm{fin}}{\to} \mathrm{StrVal}\};$$

- and a set of *functor closures* of the form $<\mathrm{X}, \mathcal{E}, \mathrm{s}>$.

We let $\mathcal{E}$ range over dynamic environments mapping value, structure and functor identifiers to core values, structure values and functor closures.

We can now present the dynamic semantics by defining evaluation judgements relating expressions to their values (Figure 10).

$$\boxed{\mathcal{E} \vdash \mathrm{e} \downarrow \mathrm{v}}$$

$$\vdots$$

$$\frac{\mathcal{E} \vdash \mathrm{s} \downarrow \mathrm{V}}{\mathcal{E} \vdash \textbf{pack}\ \mathrm{s}\ \textbf{as}\ \mathrm{S} \downarrow \mathrm{<V>}}$$

$$\frac{\mathcal{E} \vdash \mathrm{e} \downarrow \mathrm{<V>} \quad \mathcal{E}[\mathrm{X} = \mathrm{V}] \vdash \mathrm{e}' \downarrow \mathrm{v}}{\mathcal{E} \vdash \textbf{open}\ \mathrm{e}\ \textbf{as}\ \mathrm{X} : \mathrm{S}\ \textbf{in}\ \mathrm{e}' \downarrow \mathrm{v}}$$

$$\vdots$$

$$\boxed{\mathcal{E} \vdash \mathrm{b} \downarrow \mathrm{V}}$$

$$\vdots$$

$$\boxed{\mathcal{E} \vdash \mathrm{s} \downarrow \mathrm{V}}$$

$$\vdots$$

$$\frac{\mathcal{E}(\mathrm{F}) = \mathrm{<X}, \mathcal{E}', \mathrm{s}'\mathrm{>} \quad \mathcal{E} \vdash \mathrm{s} \downarrow \mathrm{V} \quad \mathcal{E}'[\mathrm{X} = \mathrm{V}] \vdash \mathrm{s}' \downarrow \mathrm{V}'}{\mathcal{E} \vdash \mathrm{F}(\mathrm{s}) \downarrow \mathrm{V}'}$$

$$\vdots$$

Fig. 10. Evaluation Judgements

To prove that the static semantics is sound for the dynamic semantics, we introduce additional classification judgements relating values to their (semantic) types (Figure 11).

We say that an environment $\mathcal{E}$ has type $\mathcal{C}$, written $\vdash \mathcal{E} : \mathcal{C}$, if, and only if, every value, structure and functor identifier declared with a type in the context $\mathcal{C}$ is assigned a value in $\mathcal{E}$ that inhabits this type.

The type soundness property can then be stated as:

Property 1 (Type Soundness)

- $\mathcal{C} \vdash \mathrm{e} : u \supset \vdash \mathcal{E} : \mathcal{C} \supset \mathcal{E} \vdash \mathrm{e} \downarrow \mathrm{v} \supset \vdash \mathrm{v} : u.$
- $\mathcal{C} \vdash \mathrm{b} : \mathcal{X} \supset \vdash \mathcal{E} : \mathcal{C} \supset \mathcal{E} \vdash \mathrm{b} \downarrow \mathrm{V} \supset \vdash \mathrm{V} : \mathcal{X}.$
- $\mathcal{C} \vdash \mathrm{s} : \mathcal{X} \supset \vdash \mathcal{E} : \mathcal{C} \supset \mathcal{E} \vdash \mathrm{s} \downarrow \mathrm{V} \supset \vdash \mathrm{V} : \mathcal{X}.$

Proof 1 (Sketch) *We need to prove the stronger properties:*

- $\mathcal{C} \vdash \mathrm{e} : u \supset \forall \psi, \mathcal{E}, \mathrm{v}.\ \vdash \mathcal{E} : \psi(\mathcal{C}) \supset \mathcal{E} \vdash \mathrm{e} \downarrow \mathrm{v} \supset \vdash \mathrm{v} : \psi(u),$
- $\mathcal{C} \vdash \mathrm{b} : \mathcal{X} \supset \forall \psi, \mathcal{E}, \mathrm{V}.\ \vdash \mathcal{E} : \psi(\mathcal{C}) \supset \mathcal{E} \vdash \mathrm{b} \downarrow \mathrm{V} \supset \vdash \mathrm{V} : \psi(\mathcal{X}),$
- $\mathcal{C} \vdash \mathrm{s} : \mathcal{X} \supset \forall \psi, \mathcal{E}, \mathrm{V}.\ \vdash \mathcal{E} : \psi(\mathcal{C}) \supset \mathcal{E} \vdash \mathrm{s} \downarrow \mathrm{V} \supset \vdash \mathrm{V} : \psi(\mathcal{X})$

which can be proved by simultaneous induction on the classification judgements.

Here, ψ is a realisation of type variables. Quantifying over all ψ allows us to prove that the rules introducing and eliminating type polymorphism are sound. To give an indication of how the proof proceeds, we will give the proof of type soundness for Rule (3). The proof remains a sketch because we do not verify the other cases, nor have we formalised the machinery necessary to do so.

Rule 3 *By induction we may assume:*

$$\forall \psi, \mathcal{E}, \mathrm{v}. \vdash \mathcal{E} : \psi(\mathcal{C}) \supset \mathcal{E} \vdash \mathrm{e} \downarrow \mathrm{v} \supset \vdash \mathrm{v} : \psi(<\exists P.\mathcal{S}>), \tag{4}$$

$\boxed{\vdash \mathrm{v} : u}$

$$\dfrac{\vdots \atop \vdash \mathrm{V} : \mathcal{X}}{\vdash <\mathrm{V}> : <\mathcal{X}>} \atop \vdots$$

$\boxed{\vdash \mathrm{V} : \mathcal{S}}$

$$\frac{\begin{array}{l}\mathcal{D}(\mathrm{V}) \supseteq \mathcal{D}(\mathcal{S}) \\ \forall \mathrm{x} \in \mathcal{D}(\mathcal{S}). \quad \vdash \mathcal{E}(\mathrm{x}) : \mathcal{S}(\mathrm{x}) \\ \forall \mathrm{X} \in \mathcal{D}(\mathcal{S}). \quad \vdash \mathcal{E}(\mathrm{X}) : \mathcal{S}(\mathrm{X})\end{array}}{\vdash \mathrm{V} : \mathcal{S}}$$

$\boxed{\vdash \mathrm{V} : \mathcal{X}}$

$$\frac{\mathcal{D}(\varphi) = P \quad \vdash \mathrm{V} : \varphi\,(\mathcal{S})}{\vdash \mathrm{V} : \exists P.\mathcal{S}}$$

$\boxed{\vdash <\mathrm{X}, \mathcal{E}, \mathrm{s}> : \mathcal{F}}$

$$\frac{\forall \varphi.\ \ \mathcal{D}(\varphi) = P \supset \forall \mathrm{V}.\ \vdash \mathrm{V} : \varphi\,(\mathcal{S}) \supset \forall \mathrm{V}'.\ \ \mathcal{E}[\mathrm{X} = \mathrm{V}] \vdash \mathrm{s} \downarrow \mathrm{V}' \supset \vdash \mathrm{V}' : \varphi\,(\mathcal{X})}{\vdash <\mathrm{X}, \mathcal{E}, \mathrm{s}> : \forall P.\mathcal{S} \to \mathcal{X}}$$

Fig. 11. Classification Judgements for Values

$$\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S}, \tag{5}$$

$$P \cap \mathcal{V}(\mathcal{C}) = \emptyset, \tag{6}$$

$$\forall \psi, \mathcal{E}, \mathrm{v}. \vdash \mathcal{E} : \psi\,(\mathcal{C}, \mathrm{X} : \mathcal{S}) \supset \ \mathcal{E} \vdash \mathrm{e}' \downarrow \mathrm{v} \supset \ \vdash \mathrm{v} : \psi\,(u)\,, \tag{7}$$

$$P \cap \mathcal{V}(u) = \emptyset. \tag{8}$$

We need to show:

$$\forall \psi, \mathcal{E}, \mathrm{v}. \vdash \mathcal{E} : \psi\,(\mathcal{C}) \supset \ \mathcal{E} \vdash \textbf{open}\ \mathrm{e}\ \textbf{as}\ \mathrm{X} : \mathrm{S}\ \textbf{in}\ \mathrm{e}' \downarrow \mathrm{v} \supset \ \vdash \mathrm{v} : \psi\,(u)\,.$$

Consider arbitrary ψ, $\mathcal{E}$ *and* v *such that:*

$$\vdash \mathcal{E} : \psi\,(\mathcal{C}), \tag{9}$$

$$\mathcal{E} \vdash \textbf{open}\ \mathrm{e}\ \textbf{as}\ \mathrm{X} : \mathrm{S}\ \textbf{in}\ \mathrm{e}' \downarrow \mathrm{v}. \tag{10}$$

It remains to show $\vdash \mathrm{v} : \psi\,(u)$.
W.l.o.g. we can assume:

$$P \cap \mathcal{V}(\psi) = \emptyset. \tag{11}$$

Inverting 10 by the evaluation rule for **open** e **as** X : S **in** e′ *we must have, for some structure value* V*:*

$$\mathcal{E} \vdash \mathrm{e} \downarrow <\mathrm{V}>, \tag{12}$$

$$\mathcal{E}[\mathrm{X} = \mathrm{V}] \vdash \mathrm{e}' \downarrow \mathrm{v}. \tag{13}$$

By induction hypothesis 4 applied to ψ, $\mathcal{E}$, $<\mathrm{V}>$, *9 and 12 we obtain:*

$$\vdash <\mathrm{V}> : \psi\,(<\exists P.\mathcal{S}>), \tag{14}$$

which, by assumption 11, may be re-expressed as:

$$\vdash <\mathrm{V}> : <\exists P.\psi\,(\mathcal{S})>. \tag{15}$$

Inverting 15, by the rule relating structure values to existential structures, we must have some realisation φ *such that:*

$$\mathcal{D}(\varphi) = P, \tag{16}$$

$$\vdash \mathrm{V} : \varphi\,(\psi\,(\mathcal{S})). \tag{17}$$

Let $\psi' = \psi \cup \varphi$*. Then, by 16, 6 and 11, we have:*

$$\psi'\,(\mathcal{C}) = \psi\,(\mathcal{C})\,. \tag{18}$$

Moreover, by 16 and 11 we have:

$$\psi'\,(\mathcal{S}) = \varphi\,(\psi\,(\mathcal{S}))\,. \tag{19}$$

Combining 9 and 17 we can show:

$$\vdash \mathcal{E}[\mathrm{X} = \mathrm{V}] : (\psi\,(\mathcal{C})), \mathrm{X} : \varphi\,(\psi\,(\mathcal{S})),$$

which, by 18 and 19, may be expressed as:

$$\vdash \mathcal{E}[\mathrm{X} = \mathrm{V}] : \psi'\,(\mathcal{C}, \mathrm{X} : \mathcal{S}). \tag{20}$$

By induction hypothesis 7 on ψ', $\mathcal{E}[\mathrm{X} = \mathrm{V}]$, v, *20 and 13 we obtain:*

$$\vdash \mathrm{v} : \psi'\,(u). \tag{21}$$

Now $\psi'\,(u) = \varphi\,(\psi\,(u)) = \psi\,(u)$*, where the first equation follows by 16 and 11, and the second follows by 16 and 8.*
Hence we can re-express 21 as

$$\vdash \mathrm{v} : \psi\,(u),$$

which is the desired result.