The Definition of Standard ML with Packages

Andreas Rossberg
Universität des Saarlandes
rossberg@ps.uni-sb.de

April 12, 2005

1 Introduction

This document formally specifies the semantics of local modules and *packages* – dynamically typed modules that are first-class values – as an extension to the functional programming language Standard ML [4]. The language thus defined is a substantial subset of a larger extension of Standard ML, a language known as Alice ML [9, 2]. Packages are the central feature of Alice ML that enables support for *typed open programming*.

Packages are a variant of the concept of *dynamics* [5, 1, 3]. Dynamics complement static typing with isolated dynamic type checking by providing a universal type of 'dynamic values' that carry run-time type information. Values of every type can be injected into the 'dynamic' type, projection is a type-matching operation that dispatches on the run-time type found in the dynamic value.

Instead of simple values, packages contain *modules*. Projection simply matches the runtime *package signature* against a static one – with full respect for subtyping. See [9] for a more detailed discussion of packages and their motivation and application in the context of Alice ML, including semantic implications and their relation to first-class modules [10].

The language specification we provide in this document is not self-enclosed. Instead, we describe the respective modifications and extensions to the Definition of Standard ML [4] that are necessary to incorporate the aforementioned constructs. Hence, the presentation is rather technical and requires a thorough understanding of the Definition itself.

We make three main modifications to the Definition:

- the distinction between core and structure level declarations has to be removed,
- type declarations need to be reflected in the dynamic semantics,
- `pack` and `unpack` expressions and package values have to be added.

Notice that the first point implies that part of SML's stratification between the core and the module language is abolished – core expressions can now contain structure declarations. However, the stratification is still maintained on the type level (which is also true for full Alice ML, since it does not provide modules as first-class values).

The second point implements the semantics of dynamic types necessary to support the dynamic type check performed when a package is unpacked. It implements a dynamically generative semantics for type abstraction and datatypes in order to maintain abstraction safety [8]. Note that we need no dynamic semantics for polymorphic type variables.

Packages themselves are straightforward given the previous two modifications.

The semantics specified in this document has been affirmed by implementing it one-to-one in the framework of HaMLet [7], a model implementation of the Definition of Standard ML. A practical implementation is available in form of the Alice System [2].

2 Changes to the Syntax of the Core (Chapter 2)

2.1 Reserved Words (Section 2.1)

The word `pack` is added to the list of reserved words used in the Core.

2.2 Grammar (Section 2.8)

Packing is added as a new expressions form:

exp	::=	...	(as before)
		`pack` *longstrid* : *sigid*	packing

Furthermore, structure declarations are allowed as core declarations:

dec	::=	...	(as before)
		strdec	structure declaration

3 Changes to the Syntax of Modules (Chapter 3)

3.1 Reserved Words (Section 3.1)

The word `unpack` is added as another additional reserved word used in Modules.

3.2 Infixed Operators (Section 3.3)

The second phrase concerning `local` is removed from the enumeration, since it is no longer part of the module language (see next section).

3.3 Grammar for Modules (Section 3.4)

Structure declarations can appear as part of plain declarations. The phrase class StrDec for structure-level declarations can hence be simplified to represent structure declarations only, and its grammar no longer needs to duplicate parts of *dec*:

$$\textit{strdec} \quad ::= \quad \texttt{structure}\ \textit{strbind}$$

Unpacking is added as a new structure expression, and former occurences of *strdec* are replaced by plain *dec*:

strexp	::=	`struct` *dec* `end`	basic
		...	(as before)
		`let` *dec* `in` *strexp* `end`	local declaration
		`unpack` *longvid* `:` *sigexp*	unpacking

Likewise, the occurence of *strdec* in toplevel declarations has to be replaced:

topdec	::=	*dec* ⟨*topdec*⟩	declaration
		...	(as before)

4 Changes to the Static Semantics for the Core (Section 4)

4.1 Compound Objects (Section 4.2)

To support local structure expressions, contexts have to include functor and signature environments:

$$C \text{ or } T, U, F, G, E \quad \in \quad \text{Context} = \text{TyNameSet} \times \text{TyVarSet} \times \text{FunEnv} \times \text{SigEnv} \times \text{Env}$$

Functor environments (FunEnv) and signature environments (SigEnv) are defined in Section 5.1 of the Definition.

4.2 Projection, Injection and Modification (Section 4.2)

The example of context modification is adapted to the extended definition of contexts, i.e. $C + (T, VE)$ means

$$(\ (T \text{ of } C) \cup T,\ U \text{ of } C,\ F \text{ of } C,\ G \text{ of } C,\ (E \text{ of } C) + VE\ )$$

4.3 Inference Rules (Section 4.10)

Expressions $\boxed{C \vdash exp \Rightarrow \tau}$

A new rule for packing expressions is added. To describe matching, it uses the same side conditions as rule 52 (Section 5.7 of the Definition), which deals with signature constraints:

$$\frac{C(longstrid) = E \qquad C(sigid) = \Sigma \qquad \Sigma \geq E' \prec E}{C \vdash \texttt{pack}\ longstrid\ :\ sigid \Rightarrow \texttt{package}} \tag{12a}$$

Comment: Signature matching $\Sigma \geq E' \prec E$ is defined in Section 5.6 of the Definition. Note that unlike rule 53, no side condition on the type names in the signature is necessary, since they cannot appear in the result type.

Declarations $\boxed{C \vdash dec \Rightarrow E}$

A new rule is added for structure declarations:

$$\frac{B \text{ of } C \vdash strdec \Rightarrow E}{C \vdash strdec \Rightarrow E} \tag{24a}$$

Comment: The projection B of C is defined in the static semantics for Modules below.

5 Changes to the Static Semantics for Modules (Section 5)

5.1 Semantic Objects (Section 5.1)

The projection C of B is adapted to the extended definition of contexts (Section 4.1).[1] It denotes the context $(T \text{ of } B, \emptyset, F \text{ of } B, G \text{ of } B, E \text{ of } B)$. Moreover, the inverse projection B of C is defined as $(T \text{ of } C, F \text{ of } C, G \text{ of } C, E \text{ of } C)$.

5.2 Inference Rules (Section 5.7)

The last paragraph of the Theorem is adapted to the extended definition of contexts by replacing the inferred sentence with $T, U, F, G, E \vdash phrase \Rightarrow A$.

[1] With the extension of contexts, a basis is turned into context rather by injection, so the notation B in Context would be more appropriate. However, we chose to keep the notation used in the Definition to avoid changing too many rules.

Structure Expressions $\boxed{B \vdash \mathit{strexp} \Rightarrow E}$

Structures and `let` prefixes contain core declarations, hence the respective rules are adapted:

$$\frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E}{B \vdash \texttt{struct}\ \mathit{dec}\ \texttt{end} \Rightarrow E} \tag{50}$$

$$\frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{strexp} \Rightarrow E_2}{B \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \Rightarrow E_2} \tag{55}$$

Unpacking requires a new rule:

$$\frac{B(\mathit{longvid}) = (\texttt{package}, \texttt{v}) \qquad B \vdash \mathit{sigexp} \Rightarrow (T)E}{B \vdash \texttt{unpack}\ \mathit{longvid}\ :\ \mathit{sigexp} \Rightarrow E} \tag{55a}$$

Comment: The side condition in rule 65 ensures that the type names in T are fresh, and hence the signature is treated as opaque.

Structure Declarations $\boxed{B \vdash \mathit{strdec} \Rightarrow E}$

Rules 56 and 58–60 are deleted, because the StrDec grammar has been simplified (see Section 3.3).

Top-level Declarations $\boxed{B \vdash \mathit{topdec} \Rightarrow B'}$

The former rule for structure declarations is adapted to the respective syntax change (see Section 3.3):

$$\frac{\begin{array}{c} C \text{ of } B \vdash \mathit{dec} \Rightarrow E \qquad \langle B \oplus E \vdash \mathit{topdec} \Rightarrow B' \rangle \\ B'' = (\text{tynames}\, E, E) \text{ in Basis}\langle + B' \rangle \qquad \text{tyvars}\, B'' = \emptyset \end{array}}{B \vdash \mathit{dec}\ \langle \mathit{topdec} \rangle \Rightarrow B''} \tag{87}$$

6 Changes to the Dynamic Semantics for the Core (Chapter 6)

6.1 Reduced Syntax (Section 6.1)

Since types have to be represented in the dynamic semantics in order to define dynamic typing, no transformations are made to delete any type declarations from the syntax. The only transformation is the removal of explicit type ascriptions "$:\ ty$".

6.2 Compound Objects (Section 6.3)

The most notable extension to the core semantics is the addition of package values, which are pairs of an environment representing the packaged structure and its interface:

$$\begin{array}{rcl} v & \in & \text{Val} = \cdots \cup \text{PackageVal} \\ (E, I) & \in & \text{PackageVal} = \text{Env} \times \text{Int} \end{array}$$

Interfaces represent signatures in the dynamic semantics. They are defined by the Definition in the Dynamic Semantics for Modules (Chapter 7), but need to be modified (see Section 7.2).

To model dynamic typing, dynamic type environments not only carry constructor environments as in the Definition, but also the corresponding type functions. That is, we introduce type structures, like they exist in the static semantics, into the dynamic semantics:

$$\begin{array}{rcl} TE & \in & \text{TyEnv} = \text{TyCon} \xrightarrow{\text{fin}} \text{TyStr} \\ (\theta,\, VI) & \in & \text{TyStr} = \text{TypeFcn} \times \text{ValInt} \end{array}$$

To be able to check equivalence of datatypes, a type structure's constructor environment is not a value environment, but instead a value interface. The Definition defines value interfaces in the Dynamic Semantics for Modules (Chapter 7), but for Alice ML they are extended to contain full type information (see Section 7.2).

Type generativity is reflected in the dynamic semantics. The state is thus extended to record the set of type names already generated:

$$(mem,\, ens,\, tns) \text{ or } s \quad \in \quad \text{State} = \text{Mem} \times \text{ExNameSet} \times \text{TyNameSet}$$

In order to support local structures, functor and signature environments have to be available within core expressions. Like in the static semantics for the core, a context is hence used:

$$C \text{ or } (F, G, E) \quad \in \quad \text{Context} = \text{FunEnv} \times \text{SigEnv} \times \text{Env}$$

Note that this definition of a dynamic context is equivalent to that of a dynamic basis (Section 7.2 of the Definition). They are hence interchangable. We define C in Basis to mean the reinterpretation of C as a basis. Vice versa, C of B denotes the reinterpretation of B as a context.

Accordingly, we have to change the definition of function closures to contain contexts:

$$(match, C,\, VE) \quad \in \quad \text{FcnClosure} = \text{Match} \times \text{Context} \times \text{ValEnv}$$

6.3 Basic Exceptions (Section 6.5)

The set BasExName of basic exceptions is extended to include the exception name denoted by the identifier `Unpack`. The exception `Unpack` is raised upon failure to unpack a package, i.e. when a dynamic type mismatch occurs.

6.4 Inference Rules (Section 6.7)

In all rules of the dynamic semantics, occurences of environment variables E on the left-hand side of the turnstile have to be replaced by context variables C. Since the modifications are mechanic, we will not replicate all rules here.

Expressions

$\boxed{C \vdash exp \Rightarrow v/p}$

$$\frac{C(longstrid) = E \qquad C(sigid) = \Sigma \qquad \Sigma \geq I \prec \mathrm{Inter}\, E}{C \vdash \texttt{pack}\ longstrid\ :\ sigid \Rightarrow (E \downarrow I, I), s} \tag{108a}$$

Comment: The package signature is transparent, i.e. all type names are realised. The interface I binds type constructors to the same type structures as the environment E, so that packages are consistent with respect to their choice of type names. Signature matching $\Sigma \geq E_1 \prec E_2$ is defined in Section 5.6 of the Definition and applies equally to the modified definition of interfaces. The operator Inter is defined in Section 7.2 of the Definition, but adapted to the new definition of interfaces (Section 7.2).

Declarations

$\boxed{C \vdash dec \Rightarrow E'/p}$

Dynamic typing implies that types have to be passed during evaluation. In Alice ML, only explicitly declared types have to be represented in the dynamic semantics. This is modelled by modifying the rules dealing with type declarations. The modified rules essentially mimic the corresponding rules in the static semantics. Note that this requires the rules for type, datatype, constructor, and exception bindings to be formulated relative to a dynamic context C.

$$\frac{C \vdash typbind \Rightarrow TE}{C \vdash \texttt{type}\ typbind \Rightarrow TE\ \mathrm{in\ Env}} \tag{115}$$

$$\frac{C + TE \vdash \mathit{datbind} \Rightarrow VE, TE \qquad TE \text{ maximises equality}}{C \vdash \texttt{datatype}\ \mathit{datbind} \Rightarrow (VE, TE) \text{ in Env}} \tag{116}$$

$$\frac{\begin{array}{c} C(\mathit{longtycon}) = (\theta, VI) \qquad TE = \{\mathit{tycon} \mapsto (\theta, VI)\} \\ VE = \{\mathit{vid} \mapsto (\mathit{vid}, \mathit{is})\ ;\ VI(\mathit{vid}) = (\sigma, \mathit{is})\} \end{array}}{C \vdash \texttt{datatype}\ \mathit{tycon}\ \texttt{=}\ \texttt{datatype}\ \mathit{longtycon} \Rightarrow (VE, TE) \text{ in Env}} \tag{117}$$

$$\frac{\begin{array}{c} C + TE \vdash \mathit{datbind} \Rightarrow VE, TE \qquad TE \text{ maximises equality} \\ C + (VE, TE) \vdash \mathit{dec} \Rightarrow E' \end{array}}{C \vdash \texttt{abstype}\ \mathit{datbind}\ \texttt{with}\ \mathit{dec}\ \texttt{end} \Rightarrow E'} \tag{118}$$

Comment: (116),(118) The freshness of type names generated by *datbind* is ensured by choosing them with respect to the state in rule 128.

A new rule is added for structure declarations:

$$\frac{C \text{ in Basis} \vdash \mathit{strdec} \Rightarrow E}{C \vdash \mathit{strdec} \Rightarrow E} \tag{123a}$$

Type Bindings $\boxed{C \vdash \mathit{typbind} \Rightarrow TE}$

$$\frac{\mathit{tyvarseq} = \alpha^{(k)} \qquad C \vdash \mathit{ty} \Rightarrow \tau \qquad \langle C \vdash \mathit{typbind} \Rightarrow TE \rangle}{\begin{array}{c} C \vdash \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle \texttt{and}\ \mathit{typbind} \rangle \Rightarrow \\ \{\mathit{tycon} \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\} \langle + TE \rangle \end{array}} \tag{127}$$

Datatype Bindings $\boxed{C \vdash \mathit{datbind} \Rightarrow VE, TE}$

$$\frac{\begin{array}{c} \mathit{tyvarseq} = \alpha^{(k)} \qquad s, C, \alpha^{(k)} t \vdash \mathit{conbind} \Rightarrow VI, s \qquad \text{arity}\, t = k \\ t \notin \mathit{tns} \text{ of } s \qquad s' = s + \{t\} \qquad \langle s', C \vdash \mathit{datbind} \Rightarrow (VE', TE'), s'' \rangle \\ VE = \{\mathit{vid} \mapsto (\mathit{vid}, \mathit{is})\ ;\ VI(\mathit{vid}) = (\sigma, \mathit{is})\} \end{array}}{\begin{array}{c} s, C \vdash \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind} \rangle \Rightarrow \\ VE \langle + VE' \rangle, \{\mathit{tycon} \mapsto (t, \text{Clos}\ VI)\} \langle + TE' \rangle, s' \langle ' \rangle \end{array}} \tag{128}$$

Comment: A datatype binding generates new type names.

Constructor Bindings $\boxed{C, \tau \vdash \mathit{conbind} \Rightarrow VI}$

$$\frac{\langle C \vdash \mathit{ty} \Rightarrow \tau' \rangle \qquad \langle \langle C, \tau \vdash \mathit{conbind} \Rightarrow VI \rangle \rangle}{\begin{array}{c} C, \tau \vdash \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle \langle \texttt{|}\ \mathit{conbind} \rangle \rangle \Rightarrow \\ \{\mathit{vid} \mapsto (\tau, \texttt{c})\} \langle + \{\mathit{vid} \mapsto (\tau' \to \tau, \texttt{c})\} \rangle \langle \langle + VI \rangle \rangle \end{array}} \tag{129}$$

Type Expressions $\boxed{C \vdash ty \Rightarrow \tau}$

Type Rows $\boxed{C \vdash tyrow \Rightarrow \varrho}$

Dynamic types are represented by the same objects as static types. Consequently, the evaluation rules (149a)-(149f) for types are identical to the rules (44)-(49) of the static semantics, except for rule 46, where the (unused) value environment meta variable VE is replaced by a variable VI ranging over value interfaces, respectively.

7 Changes to the Dynamic Semantics for Modules (Chapter 7)

7.1 Reduced Syntax (Section 7.1)

Signatures need to be fully represented during evaluation. Hence no transformations are made to the syntax.

7.2 Compound Objects (Section 7.2)

In order to represent dynamic signatures accurately, they must be represented by the same objects as in the static semantics. Interfaces are hence modified to be equivalent to environments of the static semantics:

$$\begin{aligned} I \text{ or } (SI, TI, VI) &\in \text{Int} = \text{StrInt} \times \text{TyInt} \times \text{ValInt} \\ SI &\in \text{StrInt} = \text{StrId} \xrightarrow{\text{fin}} \text{Int} \\ TI &\in \text{TyInt} = \text{TyCon} \xrightarrow{\text{fin}} \text{TyStr} \\ VI &\in \text{ValInt} = \text{VId} \xrightarrow{\text{fin}} \text{TypeScheme} \times \text{IdStatus} \end{aligned}$$

Each interface object is isomorphic to the respective environment object of the static semantics. Moreover, the definition of dynamic type structures TyStr (see Section 6.2) is isomorphic to static type structures.

The operation Inter is adapted to the changed definition of interfaces. It assumes fully polymorphic types in value environments:

$$\text{Inter}(VE) = \{vid \mapsto (\forall\alpha.\alpha, is)\,;\ VE(vid) = (v, is)\}$$

Note that this operation is used in one place only, namely rule 108a, where this assumption is safe. The extension of Inter on environments is modified such that type environments are left unmodified:

$$TI = TE$$

We need an inverse operation that injects the type parts of an interfaces into an environment:

$$\begin{array}{rcl} \mathrm{Envir}(SI,\, TI,\, VI) & = & (\mathrm{Envir}(SI),\, TI,\, \{\}) \\ \mathrm{Envir}(SI) & = & \{vid \mapsto (\forall\alpha.\alpha,\, is)\,;\ VE(vid) = (v,\, is)\} \end{array}$$

Signature environments are also equivalent to the static semantics:

$$\begin{array}{rcl} G & \in & \mathrm{SigEnv} = \mathrm{SigId} \xrightarrow{\mathrm{fin}} \mathrm{Sig} \\ \Sigma \text{ or } (T)I & \in & \mathrm{Sig} = \mathrm{TyNameSet} \times \mathrm{Int} \end{array}$$

The set IntBasis of interface bases IB is no longer needed and thus deleted. Accordingly, the definition of the operation Inter on a basis is removed. In the inference rules all occurences of the meta variable IB are replaced by B. All existing applications of Inter are removed.

The definition of the cutdown operation $\downarrow$ is modified to be consistent with the new definition of interfaces:

$$\begin{array}{rcl} SE \downarrow SI & = & \{strid \mapsto E \downarrow I\ ;\ SE(strid) = E \text{ and } SI(strid) = I\} \\ TE \downarrow TI & = & \{tycon \mapsto (\theta,\, VI)\ ;\ TE(tycon) = (\theta,\, VI) \text{ and } TI(tycon) = (\theta',\, VI')\} \\ VE \downarrow VI & = & \{vid \mapsto (v,\, is)\ ;\ VE(vid) = (v,\, is') \text{ and } VI(vid) = (\sigma,\, is)\} \end{array}$$

Note that constructor environments never need to be cut down, since they can only contain identifiers with status `c` and need to have the same domain for compatible types.

To model the dynamic semantics of opaque signature constraints an opaque cutdown operation $\Downarrow$ is defined:

$$\begin{array}{rcl} SE \Downarrow SI & = & \{strid \mapsto E \Downarrow I\ ;\ SE(strid) = E \text{ and } SI(strid) = I\} \\ TE \Downarrow TI & = & \{tycon \mapsto (\theta,\, VI)\ ;\ TE(tycon) = (\theta',\, VI') \text{ and } TI(tycon) = (\theta,\, VI)\} \\ VE \Downarrow VI & = & VE \downarrow VI \end{array}$$

Compared to the transparent cutdown operation $\downarrow$, opaque cutdown has a reversed effect on type environments: instead of the original type structures, the resulting environment will contain the type structures from the interface. Type constructors that map to an abstract type name in the interface will hence map to the same type name in the resulting environment. Along with appropriate freshness conditions on interface type, this implements dynamic generativity for opaque signature constraints. Opaque cutdown is lifted to full environments:

$$(SE,\, TE,\, VE) \Downarrow (SI,\, TI,\, VI) \quad = \quad (SE \Downarrow SI,\, TE \Downarrow TI,\, VE \Downarrow VI)$$

7.3 Inference Rules (Section 7.3)

Structure Expressions

$\boxed{B \vdash \mathit{strexp} \Rightarrow E/p}$

The rules for structures and `let` are adapted to the syntax changes arising from the merge of *dec* and *strdec* (see Section 2.2). The dynamic semantics of opaque signature constraints differs from that of transparent ones — it creates new type names:

$$\frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E}{B \vdash \texttt{struct}\ \mathit{dec}\ \texttt{end} \Rightarrow E} \tag{150}$$

$$\frac{B \vdash \mathit{strexp} \Rightarrow E \qquad B \vdash \mathit{sigexp} \Rightarrow T, I}{B \vdash \mathit{strexp}\ :\ \mathit{sigexp} \Rightarrow E \downarrow I} \tag{152}$$

$$\frac{B \vdash \mathit{strexp} \Rightarrow E \qquad B \vdash \mathit{sigexp} \Rightarrow T, I}{B \vdash \mathit{strexp}\ \texttt{:>}\ \mathit{sigexp} \Rightarrow E \Downarrow I} \tag{153}$$

$$\frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{strexp} \Rightarrow E_2}{B \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \Rightarrow E_2} \tag{155}$$

Comment: (153) Freshness of the names T in I is guaranteed by the rules for signature expressions. Opaque cutdown propagates them into the environment representing the structure, making it carry fresh abstract types.

Unpacking is modelled by two straightforward rules:

$$\frac{B(\mathit{longvid}) = ((E', I'), \mathtt{v}) \qquad B \vdash \mathit{sigexp} \Rightarrow T, I \qquad (T)I \geq I'' \prec I'}{B \vdash \texttt{unpack}\ \mathit{longvid}\ :\ \mathit{sigexp} \Rightarrow E' \downarrow I''} \tag{155a}$$

$$\frac{\begin{array}{c} B(\mathit{longvid}) = ((E', I'), \mathtt{v}) \qquad B \vdash \mathit{sigexp} \Rightarrow T, I \\ \text{there is no } I'', \text{ such that } (T)I \geq I'' \prec I' \end{array}}{B \vdash \texttt{unpack}\ \mathit{longvid}\ :\ \mathit{sigexp} \Rightarrow [\texttt{Unpack}]} \tag{155b}$$

Comment: Matching $\Sigma \geq I'' \prec I'$ is defined by the static semantics. Unpacking behaves like transparent signature constraints (cf. rule 152).

Structure Declarations

$\boxed{B \vdash \mathit{strdec} \Rightarrow E/p}$

Rules 156 and 158–160 are deleted, because the StrDec grammar has been simplified (see Section 3.3).

Signature Expressions

$\boxed{B \vdash \mathit{sigexp} \Rightarrow T, I}$

Like types, signatures have to be fully represented in the dynamic semantics. The evaluation rules for signature expressions and specifications are extended to evaluate according

to the extended definition of interface objects, which is equivalent to environment objects in the static semantics. The evaluation rules hence mirror the elaboration rules from the static semantics, except for the treatment of freshness, which is coped with by state in order to make it deterministic. The set of generated type names is inferred explicitly. It always contains fresh names:

$$\frac{B \vdash \mathit{spec} \Rightarrow T, I}{B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow T, I} \tag{162}$$

$$\frac{B(\mathit{sigid}) = (T)I \qquad T \cap \mathit{tns}\ \text{of}\ s = \emptyset}{s, B \vdash \mathit{sigid} \Rightarrow T, I, s + T} \tag{163}$$

Comment: (163) A named signature is instantiated with a fresh set of type names.

In particular, we have to deal with `where` qualifications, the same way as in the static semantics:

$$\frac{\begin{array}{ccc} B \vdash \mathit{sigexp} \Rightarrow T, I & \mathit{tyvarseq} = \alpha^{(k)} & B \vdash \mathit{ty} \Rightarrow \tau \\ I(\mathit{longtycon}) = (t, VI) & t \in T & \varphi = \{t \mapsto \Lambda\alpha^{(k)}.\tau\} \end{array}}{B \vdash \mathit{sigexp}\ \texttt{where type}\ \mathit{tyvarseq}\ \mathit{longtycon}\ \texttt{=}\ \mathit{ty} \Rightarrow T \setminus \{t\}, \varphi(I)} \tag{163a}$$

Comment: Well-formedness of φ and admissibility with respect to equality is ensured by the static semantics.

Signature Bindings

$\boxed{B \vdash \mathit{sigbind} \Rightarrow G}$

Only signature bindings require closed signatures, so closure is built into the rule:

$$\frac{B \vdash \mathit{sigexp} \Rightarrow T, I \qquad \langle B \vdash \mathit{sigbind} \Rightarrow G \rangle}{B \vdash \mathit{sigid}\ \texttt{=}\ \mathit{sigexp}\ \langle \texttt{and}\ \mathit{sigbind} \rangle \Rightarrow \{\mathit{sigid} \mapsto (T)I\} \langle + G \rangle} \tag{165}$$

Comment: Once closed, the type names of T are subject to α-renaming.

Specifications

$\boxed{B \vdash \mathit{spec} \Rightarrow T, I}$

Rules for specifications and descriptions mimic the static semantics. However, the set of local type names is managed explicitly:

$$\frac{C\ \text{of}\ B \vdash \mathit{valdesc} \Rightarrow \mathit{VI}}{B \vdash \texttt{val}\ \mathit{valdesc} \Rightarrow \emptyset, \text{Clos}\ \mathit{VI}\ \text{in Int}} \tag{166}$$

$$\frac{\vdash \mathit{typdesc} \Rightarrow \mathit{TI} \qquad \forall(t, \mathit{VI}) \in \mathrm{Ran}\ \mathit{TI}, t \text{ does not admit equality}}{B \vdash \texttt{type}\ \mathit{typdesc} \Rightarrow \text{tynames } \mathit{TI}, \mathit{TI} \text{ in Int}} \tag{167}$$

$$\frac{\vdash \mathit{typdesc} \Rightarrow \mathit{TI} \qquad \forall(t, \mathit{VI}) \in \mathrm{Ran}\ \mathit{TI}, t \text{ admits equality}}{B \vdash \texttt{eqtype}\ \mathit{typdesc} \Rightarrow \text{tynames } \mathit{TI}, \mathit{TI} \text{ in Int}} \tag{168}$$

$$\frac{\begin{array}{c} C \text{ of } B + \mathit{TE} \vdash \mathit{datdesc} \Rightarrow \mathit{VI}, \mathit{TI} \qquad T = \{t;\ (t, \mathit{VI}') \in \mathrm{Ran}\ \mathit{TI}\} \\ \mathit{TE} = \mathit{TI} \qquad\qquad \mathit{TI} \text{ maximises equality} \end{array}}{B \vdash \texttt{datatype}\ \mathit{datdesc} \Rightarrow T, (\mathit{VI}, \mathit{TI}) \text{ in Int}} \tag{169}$$

$$\frac{B(\mathit{longtycon}) = (\theta, \mathit{VI}) \qquad\qquad \mathit{TI} = \{\mathit{tycon} \mapsto (\theta, \mathit{VI})\}}{B \vdash \texttt{datatype}\ \mathit{tycon}\ \texttt{=}\ \texttt{datatype}\ \mathit{longtycon} \Rightarrow \emptyset, (\mathit{VI}, \mathit{TI}) \text{ in Int}} \tag{170}$$

$$\frac{C \text{ of } B \vdash \mathit{exdesc} \Rightarrow \mathit{VI}}{B \vdash \texttt{exception}\ \mathit{exdesc} \Rightarrow \emptyset, \mathit{VI} \text{ in Int}} \tag{171}$$

$$\frac{B \vdash \mathit{strdesc} \Rightarrow T, \mathit{SI}}{B \vdash \texttt{structure}\ \mathit{strdesc} \Rightarrow T, \mathit{SI} \text{ in Int}} \tag{172}$$

$$\frac{B \vdash \mathit{sigexp} \Rightarrow T, I}{B \vdash \texttt{include}\ \mathit{sigexp} \Rightarrow T, I} \tag{173}$$

$$\frac{}{B \vdash \quad \Rightarrow \emptyset, \{\} \text{ in Int}} \tag{174}$$

$$\frac{B \vdash \mathit{spec}_1 \Rightarrow T_1, I_1 \qquad B + \mathrm{Envir}\, I_1 \vdash \mathit{spec}_2 \Rightarrow T_2, I_2}{B \vdash \mathit{spec}_1\ \langle;\rangle\ \mathit{spec}_2 \Rightarrow T_1 \cup T_2, I_1 + I_2} \tag{175}$$

$$\frac{\begin{array}{ll} B \vdash \mathit{spec} \Rightarrow T, I & I(\mathit{longtycon}) = (t_i, \mathit{VI}_i),\ i = 1..n \\ t \in \{t_1, \ldots, t_n\} & t \text{ admits equality, if some } t_i \text{ does} \\ T' = (T \setminus \{t_1, \ldots, t_n\}) \cup \{t\} & \varphi = \{t_1 \mapsto t, \ldots, t_n \mapsto t\} \end{array}}{B \vdash \mathit{spec}\ \texttt{sharing type}\ \mathit{longtycon}_1\ \texttt{=}\ \ldots\ \texttt{=}\ \mathit{longtycon}_n \Rightarrow T', \varphi(I)} \tag{175a}$$

Comments:
(167), (168) The type interfaces TI cannot contain any other type names than freshly generated ones.
(175) Note that T_1 and T_2 are always disjoint, because state is threaded by the state convention.
(175a) The static semantics ensures that $\{t_1, \ldots, t_n\} \subseteq T$.

Value Descriptions $\boxed{C \vdash \mathit{valdesc} \Rightarrow \mathit{VI}}$

$$\frac{C \vdash \mathit{ty} \Rightarrow \tau \qquad\qquad \langle C \vdash \mathit{valdesc} \Rightarrow \mathit{VI}\rangle}{C \vdash \mathit{vid}\ :\ \mathit{ty}\ \langle\texttt{and}\ \mathit{valdesc}\rangle \Rightarrow \{\mathit{vid} \mapsto (\tau, \mathtt{v})\}\langle + \mathit{VI}\rangle} \tag{176}$$

Type Descriptions $\boxed{\vdash typdesc \Rightarrow TI}$

$$\frac{tyvarseq = \alpha^{(k)} \qquad \text{arity}\, t = k \qquad t \notin tns\ \text{of}\ s \qquad s' = s + \{t\} \qquad \langle s' \vdash typdesc \Rightarrow TI, s'' \rangle}{s \vdash tyvarseq\ tycon\ \langle \texttt{and}\ typdesc \rangle \Rightarrow \{tycon \mapsto (t, \{\})\}\langle + TI \rangle, s'\langle' \rangle} \tag{177}$$

Comment: Type descriptions are abstract and generate new type names.

Datatype Descriptions $\boxed{C \vdash datdesc \Rightarrow VI, TI}$

$$\frac{tyvarseq = \alpha^{(k)} \qquad C, \alpha^{(k)} t \vdash condesc \Rightarrow VI \qquad \text{arity}\, t = k \qquad t \notin tns\ \text{of}\ s \qquad s' = s + \{t\} \qquad \langle s', C \vdash datdesc \Rightarrow VI', TI', s'' \rangle}{\begin{array}{l} s, C \vdash tyvarseq\ tycon\ \texttt{=}\ condesc\ \langle \texttt{and}\ datdesc \rangle \Rightarrow \\ \qquad VI\langle + VI' \rangle, \{tycon \mapsto (t,\ VI)\}\langle + TI' \rangle, s'\langle' \rangle \end{array}} \tag{178}$$

Comment: Datatypes are also generative.

Constructor Descriptions $\boxed{C, \tau \vdash condesc \Rightarrow VI}$

$$\frac{\langle C \vdash ty \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash condesc \Rightarrow VI \rangle\rangle}{\begin{array}{l} C, \tau \vdash vid\ \langle \texttt{of}\ ty \rangle\ \langle\langle \texttt{|}\ condesc \rangle\rangle \Rightarrow \\ \qquad \{vid \mapsto (\tau, \texttt{c})\}\langle + \{vid \mapsto (\tau' \to \tau, \texttt{c})\} \rangle \langle\langle + VI \rangle\rangle \end{array}} \tag{179}$$

Exception Descriptions $\boxed{C \vdash exdesc \Rightarrow VI}$

$$\frac{\langle C \vdash ty \Rightarrow \tau \rangle \qquad \langle\langle C \vdash exdesc \Rightarrow VI \rangle\rangle}{\begin{array}{l} C \vdash vid\ \langle \texttt{of}\ ty \rangle\ \langle\langle \texttt{and}\ exdesc \rangle\rangle \Rightarrow \\ \qquad \{vid \mapsto (\texttt{exn}, \texttt{e})\}\langle + \{vid \mapsto (\tau \to \texttt{exn}, \texttt{e})\} \rangle \langle\langle + VI \rangle\rangle \end{array}} \tag{180}$$

Structure Descriptions $\boxed{B \vdash strdesc \Rightarrow T, SI}$

$$\frac{B \vdash sigexp \Rightarrow T, I \qquad \langle B \vdash strdesc \Rightarrow T', SI \rangle}{B \vdash strid\ :\ sigexp\ \langle \texttt{and}\ strdesc \rangle \Rightarrow T\langle \cup T' \rangle, \{strid \mapsto I\}\langle + SI \rangle} \tag{181}$$

Comment: The type name sets are always disjoint, because state is threaded by the state convention.

Functor Bindings $\boxed{B \vdash funbind \Rightarrow F}$

The rule for functor bindings can ignore the type names of the parameter signature, since the interface I stored in the functor closure is only used for transparent cutdown (rule 154 for functor application is unchanged):

$$\frac{B \vdash sigexp \Rightarrow T, I \qquad \langle B \vdash funbind \Rightarrow F \rangle}{\begin{array}{r} B \vdash funid\ \texttt{(}\ strid\ \texttt{:}\ sigexp\ \texttt{)}\ \texttt{=}\ strexp\ \langle \texttt{and}\ funbind \rangle \Rightarrow \\ \{funid \mapsto (strid : I, strexp, B)\}\langle + F \rangle \end{array}} \tag{182}$$

Top-level Declarations $\boxed{B \vdash topdec \Rightarrow B'/p}$

The former rule for structure declarations is adapted to the respective syntax change (see Section 3.3). Note that the rule given in the Definition is incorrect [6]. We give a corrected version:

$$\frac{C \text{ of } B \vdash dec \Rightarrow E \qquad B' = E \text{ in Basis} \qquad \langle B + B' \vdash topdec \Rightarrow B'' \rangle}{B \vdash dec\ \langle topdec \rangle \Rightarrow B' \langle + B'' \rangle} \tag{184}$$

8 Changes to Derived Forms (Appendix A)

We introduce an additional expression derived form for opaque packing:

Expressions *exp*

`pack` *longstrid* `:>` *sigid*	`let structure` *strid* `=` *longstrid* `:>` *sigid* `in pack` *strid* `:` *sigid* `end`

(*strid* new)

The derived form for structure expressions has to mention *dec*. Moreover, a slightly more general syntax for `unpack` is supported (the set InfExp of infix expressions is defined in Appendix B):

Structure Expressions *strexp*

funid `(` *dec* `)`	*funid* `(` `struct` *dec* `end` `)`
`unpack` *infexp* `:` *sigexp*	`let val` *vid* `=` *infexp* `in unpack` *vid* `:` *sigid* `end`

(*vid* new)

9 Changes to the Full Grammar (Appendix B)

Packing is added to the full syntax of expressions:

exp	::=	...	(as before)
		`pack` *longstrid* `:` *sigid*	packing
		`pack` *longstrid* `:>` *sigid*	opaque packing

Structure declarations appear as core declarations:

dec	::=	...	(as before)
		strdec	structure declaration

10 Changes to The Initial Static Basis (Appendix C)

The package type is a new primitive type. It is added to the initial type name set:

$$T_0 = \{\ldots, \texttt{package}\}$$

Likewise, `package` is added to the initial type environment TE_0:

tycon	$\mapsto$	$(\ \theta$	$\{vid_1 \mapsto (\sigma_1, is_1), \ldots, vid_n \mapsto (\sigma_n, is_n)\}\ )$	$(n \geq 0)$
`package`	$\mapsto$	(`package`,	$\{\}\ )$	

The `Unpack` exception is added to the initial value environment VE_0 as a nonfix identifier:

NONFIX

vid	$\mapsto$	(σ, is)
`Unpack`	$\mapsto$	(`Unpack`, `e`)

11 Changes to The Initial Dynamic Basis (Appendix D)

The initial dynamic basis has to be adapted to the extended definition of dynamic basis (see Section 7.2) and contains the exception `Unpack` and the type `package`. That is, we have

$$VE_0 = \{\ldots, \texttt{Unpack} \mapsto (\texttt{Unpack}, \texttt{e})\}$$

and the initial dynamic type environment TE_0 is actually equivalent to the initial static type environment (cf. Section 6.2):

$$TE_0^{(DYN)} = TE_0^{(STAT)}$$

Moreover, the initial type name set tns_0 had to be defined to be equal to T_0, but the Definition does not specify the initial state anyways [6].

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[2] Alice Team. *The Alice System.* Programming System Lab, Universität des Saarlandes, `http://www.ps.un-sb.de/alice/`, 2003.

[3] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[4] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *Definition of Standard ML (Revised).* The MIT Press, 1997.

[5] Alan Mycroft. Dynamic types in ML, 1983.

[6] Andreas Rossberg. Defects in the Revised Definition of Standard ML. Technical report, Saarland University, Saarbrücken, Germany, October 2001. `http://www.ps.uni-sb.de/Papers/`.

[7] Andreas Rossberg. *HaMLet – To Be Or Not To Be Standard ML.* Programming System Lab, Universität des Saarlandes, `http://www.ps.un-sb.de/hamlet/`, 2002.

[8] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Principles and Practice of Declarative Programming*, Uppsala, Sweden, August 2003.

[9] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice ML through the looking glass. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5, Munich, Germany, 2005. Intellect.

[10] Claudio Russo. First-class structures for Standard ML. In Gert Smolka, editor, *9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, Berlin, Germany, March 2000. Springer-Verlag.