Extended ML: Language Design
Version 0.1

Don Sannella
The Common Framework Initiative

January 10, 1996

This document is part of the CFI Catalogue of Algebraic Specification Frameworks[1]. It was provided by *Don Sannella*; any comments and corrections should be addressed to `dts@dcs.ed.ac.uk`.

It will be made available formatted for printing as compressed Postscript[2] (~350Kb) and DVI[3] (~50Kb).

The WWW version provides only a rough approximation (mostly generated automatically by Hyperlatex) to the symbols used in the formatted versions. Please inform `pdmosses@brics.dk` about any places where an alternative approximation is desirable!

N.B. All sections of the catalogue are *TENTATIVE* first versions, unless explicitly marked otherwise. Sections dealing with Methodology, Tools, and Documentation will be added later.

1 Specification in the Small

1.1 Basic Constructs

1.1.1 signatures

Characterize:

Signatures contain (possibly polymorphic) type constructors, values with (possibly higher-order) (possibly polymorphic) types, exceptions with (possibly higher-order) non-polymorphic types, and substructures with their signatures. The possible presence of substructures imposes a hierarchical structure. Shared components may be declared using sharing equations. Signatures are often "non-closed" or "incomplete" in the sense that values and/or exceptions can have types that refer to type constructors taken from the context rather than from the signature itself. All this is as in Standard ML except for the lack of polymorphic exceptions.

Notation for declaring sorts/types:

`type` *typdesc* `and` ... `and` *typdesc* for an "ordinary" type.

`eqtype` *typdesc* `and` ... `and` *typdesc* for a type admitting equality.

`datatype` *datdesc* `and` ... `and` *datdesc* for a type with constructors.

Here, *typdesc* has the form `('a1,...,'an) t` and *datdesc* has the form `('a1,...,'an) t = con1` ⟨`of ty1`⟩ `|...| conm` ⟨`of tym`⟩

Notation for declaring operations/functions, predicates/relations:

`val id1:ty1 and ... and idn:tyn`. Predicates/relations are just `bool`-valued functions.

[1] http://www.brics.dk/Projects/CFI/Catalogue
[2] ftp://ftp.brics.dk/pub/BRICS/Projects/CFI/Catalogue/EML.ps.Z
[3] ftp://ftp.brics.dk/pub/BRICS/Projects/CFI/Catalogue/EML.dvi

Notation for declaring variables:

"Pattern" syntax as in Standard ML is used. Here's a simple example: `forall (x,y) => x@y==[] implies x==[]`. Types are optional.

Any restrictions on order of declarations:

Linear visibility, otherwise no restriction.

Notation for indicating local/private/hidden items:

Local specifications as in Standard ML are used: `local` ... `in` ... `end`. In some papers, a box is drawn around hidden items in specifications to distinguish them from visible items.

1.1.2 axioms

Characterize:

Boolean expressions of Standard ML, extended with constructs corresponding to logical (extensional) equality, assertions about the outcome of evaluating expressions, and quantification.

Notation:

Quantification: `forall` *pattern* `=>` *exp*, likewise for `exists`. (This is a special case; the general case is `forall` *pat1* `=>` *exp1* | ... | *patn* `=>` *expn*.)

Logical equality: *exp1*`==`*exp2*. (This is not the same as Standard ML's computational equality *exp1*`=`*exp2*.)

Termination assertion: *exp* `terminates` (and *exp* `proper` additionally asserts that the result is not an exception).

Exception-raising assertion: *exp* `raises exc`

Logical connectives: `implies` and Standard ML's `not`, `andalso`, `orelse`

1.1.3 models

Characterize the structure of a model:

A model assigns concrete (term-generated) Standard ML types to type identifiers, expressible Standard ML values to value identifiers, and models to substructure identifiers.

Do specifications have loose semantics?

Yes.

1.1.4 constraints

Characterize (initiality, (free) generating, etc.):

Free generating constraints, but without axioms on the enrichment. But the logic is powerful enough to express constraints without the need to add them "on top of" the logic.

Notation:

See 1.1.1 above. Example: `datatype t = foo1 of int | foo2 of t*bool`.

1.1.5 status

Can one indicate the status of items and axioms, e.g., differentiate between constructors/selectors, axioms/theorems, equations/rewrite-rules?

Constructors versus ordinary values: yes. Otherwise: no.

Notation:

`datatype t = foo of int | ...` (`foo` is a constructor) versus `val foo:int->t` (`foo` is an arbitrary value, possibly a constructor).

1.1.6 other

Any feature not covered above:

Datatype definitions may be recursive and general recursion can be used in value definitions.

1.2 Types

Is type-checking decidable?

Of course! And types in declarations/expressions can (almost) always be inferred automatically.

1.2.1 predefined types

Notation for predefined types (e.g., Booleans):

As in Standard ML: `bool`, `int`, `real`, `string`, `'a list`, `unit`, `exn`.

1.2.2 algebra of types

Notation for predefined type constructors (e.g., enumerations, records):

Products: `(1,true,"abc"):int*bool*string`.

Disjoint sums: If `datatype t = foo1 of int | foo2 of bool` then we have `foo1 3:t` and `foo2 true:t`. Enumerations are a special case.

Records: `{name="Fred",age=25}:{name:string,age:int}`. (Products are in fact a special case of records.)

Function spaces: `fn x=>x:'a->'a`.

1.2.3 parametrized type constructors

Notation for declaring parametrized type constructors:

See 1.1.1 above. Examples: `type 'a t` and `datatype 'a t = foo1 of 'a | foo2 of 'a t * bool`.

1.2.4 polymorphic types

How is polymorphism expressed: free variables, quantification, subtypes?

Free variables.

Notation:

`[]:'a list`, for example.

Can the effect of type classes be achieved?

Yes, but (with the exception of equality types) only using “in the large” features.

Notation:

To get a type `S.t` with an order relation `S.le:S.t*S.t->bool`:

```
signature ORD = sig type t
                    val le : t * t -> bool
                    axiom forall x => le(x,x)
                    axiom ...
                end;
structure S : ORD = ?
```

1.2.5 dependent types

Can types depend on values (e.g., lists of definite length)?
No.
Notation:
If not, can the same effect be achieved by other means?
It is possible to define a functor (see "in the large" below) taking a value `bound:int` as input and producing as output a type of lists with a `cons` function that raises an exception when an attempt is made to create a list longer than `bound`.

1.2.6 other

Any feature not covered above:

1.3 Partiality

Can partiality be represented?
Yes, functions may be partial and axioms must take this into account. Partial functions may arise because of non-termination or because an exception is raised.

1.3.1 subtypes defined by predicates

Are such 'semantic' subtypes allowed?
No.
Notation:
Is 'subtype checking' decidable?

1.3.2 subtypes declared in the signature

Notation:
Not available.
Is the interpretation of functions 'universal' or 'overloaded'?
Are 'retracts' available?

1.3.3 equations

Notation:
Logical equality: *exp1* `==` *exp2*
Computational equality: *exp1* `=` *exp2*, only well-formed if *exp1* and *exp2* are of an "equality type".
Interpretation (existential, strong, weak):
Logical equality: extensional, and reflexive for all expressions including expressions whose evaluation fails to terminate or raises an exception ("strong" equality).
Computational equality: as in Standard ML, so has no value if either expression fails to terminate ("existential" equality).

1.3.4 (un)definedness

Notation for asserting definedness:
exp `terminates` to assert termination; *exp* `proper` to additionally assert that the result is not an exception.
Notation for asserting undefinedness:
`not (`*exp* `terminates)` to assert non-termination; *exp* `raises exc` to assert that a particular exception is raised; *exp* `raises _` to assert that some exception is raised.

Canonical undefined term:

There isn't a "canonical" undefined term. Here's a term that fails to terminate: `let fun loop() = loop() in loop() end`. Here's a term that raises exception `exc`: `raise exc`.

1.3.5 logic

Notation and interpretation for logical connectives:

As in Standard ML (prefix `not`, infix `andalso` and `orelse`), with infix `implies` added. The interpretation is "sequential" as in Standard ML, e.g. `true orelse loop()` yields `true`.

If the logic is two-valued: can truth-values be undetermined/irrelevant?

The logic is not two-valued.

1.3.6 strictness

Are all operations strict w.r.t. undefinedness?

All functions are strict with respect to non-termination and exceptions. But not all term-building operations are strict: e.g. `fn x => _` is non-strict with respect to non-termination and exceptions, and `_ handle` *exc* `=>` *exp* is non-strict with respect to exceptions.

If so, how is the effect of an if-then-else operation provided?

If-then-else is not a function; it is a term-building operation.

Can exception-handling and/or error-recovery be expressed?

Yes, just as in Standard ML.

1.3.7 other

Any feature not covered above:

Quantification is over "defined" values. Defined values of function type include partial functions as well as total ones.

1.4 Notations

1.4.1 offside rule

Is layout significant for parsing?

No.

If so, explain how, e.g., 'offside rule':

1.4.2 graphical symbols

Are non-ASCII characters allowed?

No.

If so, is there always an equivalent ASCII representation?

1.4.3 overloading

Can the same declared operation symbol be used on distinct (unrelated) types?

User-defined overloaded functions are not permitted. Of course, the same name may be used for different things in different structures, so for example `A.f` and `B.f` may co-exist.

Are there any built-in overloaded operation symbols?

Yes, arithmetic operations work for both `real` and `int`.

1.4.4 mixfix notation

Can applications of declared operations/predicates be written prefix, infix, postfix, mixfix?

Prefix and infix: yes. Postfix and mixfix: no.

Notation for declaring fixity:

`infix n id` for making `id` left-associative infix with precedence `n`; `infixr n id` for making `id` left-associative infix with precedence `n`.

Notation for applications with alternative fixity, if any:

If `f` has been declared as infix, then `op f` can be used in prefix position. Example: `op +(1,2)`. Fixity is subject to scoping.

Can operations with varying numbers of arguments be declared?

No, other than via the use of lists.

Notation:

1.4.5 operator precedence

Can grouping in terms be disambiguated using precedence?

Yes.

If so, how is precedence declared/determined?

See 1.4.4 above for syntax of infix directives with precedence.

1.4.6 associativity

Can (binary) operations be declared associative?

Yes.

Can (binary) operations be declared left- or right-associative?

Yes.

Notation:

See 1.1.4 above for syntax of infix directives with left/right associativity.

1.4.7 values of predefined types

Notation:

`bool` has `true`, `false`; `int` has ... `2`, `~1`, `0`, `1`, `2`, ...; `int list` has `[1,2,3]`, `...; exn has Bind, Match, ...; etc.`

1.4.8 other

Any feature not covered above:

Expression syntax is as in Standard ML.

1.5 Abbreviations

1.5.1 type inference

Can instantiation of polymorphic operations be inferred? If so: always, usually, or only sometimes?

Yes, provided overloading can be resolved.

Are explicit 'casts' allowed?

No.

Notation:

1.5.2 conditionals

Can several conditional axioms with the same tests be combined?
Yes:

```
axiom exp implies exp1
axiom exp implies exp2
```

is equivalent to

```
axiom exp implies (exp1 andalso exp2)
```

1.5.3 variable abstraction

Can terms be assigned to variables explicitly in formulae or subterms, e.g., using let or where?
Yes, using `let`.

1.5.4 variable inference

Can/must declarations of variables be left implicit?
Variables must be bound, but their types can be left implicit.
Can variables of unrestricted type ('wild cards') be declared?
Yes, e.g. `let val x::_ = l in` ... `end`

1.5.5 lambda-abstraction

Is it allowed in terms?
Yes.
Notation:
As in Standard ML: `fn` *match*, where *match* is a list of pattern/expression pairs. For example, `fn x=>x` or `fn [] => 0 | x::l => x`.

1.5.6 other

Any feature not covered above:
Type abbreviations: `type t =` *ty*.

1.6 Other

Any aspect not covered above:

2 Specification in the Large

2.1 Structuring Concepts

2.1.1 structure of specifications v. programs

Does the specification language include a programming language?
Yes, it includes *the* programming language: Standard ML!
If so, what differences are there between specification and program structuring?
The structuring *mechanisms* are the same. But the structure of a program (text) is not required to be the same as the structure of its specification (text).

2.1.2 structuring logic/institution-independence

Are all the structuring constructs independent of the details of 'specification in the small'?

In essence, yes. There is however an interaction between signature matching and polymorphism.

2.1.3 other

Any feature not covered above:

Functors (see 2.3 below), substructures.

2.2 Basic Constructs

2.2.1 naming

What are the scopes of declared names?

As in Standard ML.

Notation for referring to particular names:

Just use the name! If it is in a substructure, then "dot notation" is used, e.g. `A.B.C.f`.

How are entities having the same name treated when combined (by union, import, etc.):

If they have the same name but are in different substructures, then they remain different (see 1.4.3 above). If they *really* have the same name, then a later import ends the scope of an earlier one.

2.2.2 union

Notation:

`include SIG1 SIG2` imports both `SIG1` and `SIG2` into the current specification.

Is the combination 'syntactic' or 'semantic':

Semantic.

2.2.3 renaming

Notation:

Renaming in program units (structures) is by declaring a new type/value and defining it using the old type/value:

```
type t2=t1
val v2=v1
```

Renaming of value constructors (except for exception constructors) is not possible.

Renaming in specifications is similar, but requires the use of sharing equations (for types) and axioms (for values):

```
type t2 sharing type t2=t1
val v2 axiom v2=v1
```

Are unrenamed entities hidden, preserved, or forbidden?

Preserved.

2.2.4 hiding

Notation:

As in Standard ML: `local spec1 in spec2 end`.

2.2.5 sub-specifications

Can parts of specifications be inherited?

Specifications can be inherited, but *parts* of specifications cannot be inherited except by using `local` or re-using names to hide the part that is not to be inherited.

If so, is multiple inheritance allowed?

Yes.

Notation:

`include SIG1 SIG2`

2.2.6 nesting of signatures

Can signatures be nested?

Yes.

Notation:

```
sig ...
    structure S:sig ... end
    ...
end
```

2.2.7 morphisms as entities

Notation:

Signature morphisms are not available as entities.

2.2.8 behavioural encapsulation

Notation:

`functor F(X:SIG1):SIG2 = body` requires structures produced by `body` to "behaviourally satisfy" `SIG2` for any structure `X` that satisfies `SIG1` "literally". (This aspect of the design of EML is not captured in the current version of the semantics.)

2.2.9 other basic structuring constructs

Notation and interpretation:

Sharing equations: e.g. `sharing A.B = C.D`, `sharing type A.t = B.s`.

2.2.10 other

Any feature not covered above:

2.3 Parametrization

2.3.1 specifications parametrized by programs

Notation:

As in Standard ML, signatures can contain free structure (i.e. program) identifiers. So e.g. in `functor F(X:SIG1):SIG2 = body`, the specification `SIG2` may refer to the structure `X`. But specifications parameterised by programs are not "first-class citizens".

2.3.2 specifications of parametrized programs

Notation:

`functor F(X:SIG1):SIG2 = ?` specifies a parameterised program called `F` mapping structures matching `SIG1` to structures matching `SIG2`.

2.3.3 higher-order parametrization

Notation:

Higher-order parameterisation in the large is not available.

2.3.4 dependent specifications

Notation:

See 2.3.1 and 2.3.2.

2.3.5 uniqueness of instantiations

Do two identical instantiation expressions denote the same entity?

No.

2.3.6 other

Any feature not covered above:

2.4 Multiple Logics

2.4.1 heterogeneous specifications

Can several different logics be used in the same specification?

No.

2.4.2 heterogeneous developments

Is it possible to change the logic during development?

No.

2.4.3 logic morphisms, representation morphisms

With heterogeneity, what constructs support the composition/change of logics?

2.4.4 navigation in logic graph

With heterogeneity, what is the relationship between the different logics provided?

2.4.5 composition, instantiation, sub-framework

Notation:

2.4.6 other

Any feature not covered above:

2.5 Other

Any aspect not covered above:

References:

S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML: a gentle introduction. To appear in *Theoretical Computer Science*. Available on WWW at `http://www.dcs.ed.ac.uk/~dts/pub/gentle-tcs.ps`.

S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML. Report ECS-LFCS-94-300, Univ. of Edinburgh (1994).

E. Kazmierczak. Modularizing the specification of a small database system in Extended ML. *Formal Aspects of Computer Science* 4(1):100-142 (1992).

D. Sannella. Formal program development in Extended ML for the working programmer. *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park. Springer Workshops in Computing, 99–130 (1991).

D. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Computer Programming*, Guildford. Springer LNCS 240, 364–389 (1986).

D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. 3rd Intl. Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 375–389 (1989).

D. Sannella and A. Tarlecki. Extended ML: past, present and future. *Proc. 7th Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 297–322 (1991).