A PROPOSAL FOR STANDARD ML

(second draft)

Second Draft

A Proposal for Standard ML

Robin Milner, June '83

1. Introduction

1.1 How the proposal evolved

Over the past few years ML has been used by several people for serious work; in parallel, HOPE has been used similarly. The original ML (on DEC-10) ~~lacked~~ was incomplete in some ways, redundant in others. Some of these faults were remedied, and valuable extensions made, by Cardelli in his VAX version; other faults could be mended by borrowing ideas from HOPE.

In April '83, prompted by Bernard Sufrin, I wrote a tentative proposal to consolidate ML, and while doing so became convinced that this consolidation was possible while still keeping its character. Many people immediately discussed the proposal (see below for a list of people; I hope it's complete); there was a wide difference of views, but also much consensus - and of course several bad things in the proposal were brought to light.

This proposal tries to represent the consensus. One point of consensus was that a good language doesn't get designed by a committee, so I have tried to see it as a whole - despite the fact that most of the good ideas that make it different from previous ML are not my ideas. The people who contributed were: Rod Burstall, Luca Cardelli, Guy Cousineau, Mike Gordon, Gérard Huet, David MacQueen, Robin Milner, Kevin Mitchell, Brian Monahan, Peter Mosses, Alan Mycroft, Larry Paulson, David Rydeheard, Don Sannella, David Schmidt, John Scott, Stefan Sokolowski.

It was extremely lucky that we managed to have several separate talks between April and June; Both Dave MacQueen and Luca Cardelli just happened to be in Europe, and Dave gave a lot of coherence by visiting Edinburgh, Cambridge and INRIA. Guy Cousineau also happened to visit Edinburgh from INRIA, and Mike Gordon and Larry Paulson came from from Cambridge to Edinburgh. Altogether, we couldn't have chosen a better time to do the job. Also, Luca Cardelli has a detailed draft of an ML manual, at present describing his VAX version; he very generously offered to freeze it until this proposal is worked out.

With all this good fortune, I hope that people will be able to accept the various ways in which this language falls short of their expectations. I am ready to act as the focus for corrections to the proposal; I think it is converging, and that a corrected version should soon appear in "Polymorphism"; during the next few months the experience of implementation may produce a few (I hope minor) amendments.

This is a good place to recall that three people – not mentioned above – worked on the design and implementation of ML originally, and the language owes a lot to them: Lockwood Morris, Malcolm Newey and Chris Wadsworth.

1.2 The purpose of the proposed language

This proposed ML is not intended to be *the* functional language; there are too many degrees of freedom for such a thing to exist (lazy or eager evaluation, presence or absence of references and assignment, types-as-parameters or polymorphic typechecking, ...). Nor is it meant to be a commercial product. Mainly, it aims to be a vehicle for research in functional language design, and a means for propagating the functional programming craft and for developing functional styles.

The first of these aims, research, demands that it should be clearly delineated, with well-defined omissions. One such omission is the notion of module (as in e.g. HOPE or MODULA 2); another is polymorphic references and assignment. Since many people want these — particularly modules — a justification is needed.

For modules, it seemed preferable to me to avoid inserting a tentative and modest form as standard; rather, it is better to leave the way clear for stronger proposals. Such a proposal, perhaps from those with experience of modules in HOPE, is likely to come up soon; it could be called MML (Modular ML), and is likely to be compatible with almost all the present proposal (in fact, all except the *abstype* construct and the *spec* directives). However, this proposal includes one thing which – though not the whole essence of the module idea – is always meant to be provided by modules, namely the possibility of precompilation of parts of programs. Several people, including MacQueen, Cardelli, Mitchell and Paulson, argue convincingly for this; it seems to be provided without much fuss by the spec directive (Section 4.3).

For polymorphic references and assignment, we do have an elegant and sound scheme worked out by Luis Damas, but it is not documented and we would do better to wait for a clear exposition, either from Damas or – as promised – from MacQueen. In the proposed language much can be done to get the polymorphic effect by passing assignment functions as parameters; it is worthwhile experimenting with this method, with the advantage of keeping the well-understood type-checking discipline, which derives from Curry's Combinatory logic via Hindley.

The second aim, teaching and propagating, will benefit from a language which is well-rounded and not too large, and which doesn't depart far from what we know works well. It is quite encouraging that the syntax of a bare form of the proposed language – without derived forms – fits onto a single page (Section 2.8), and that it is easily recognised as ML with injections of HOPE.

In Section 11 are some reports of points made in various discussions between the Tentative proposal (April) and now.

1.3 Principles followed in this proposal

The overriding principle of this design is that the language should be restricted to ideas which are well-tried, either in previous versions of ML or in other functional languages (in fact, the main other source is HOPE, mainly for its pattern-matching of arguments). A second principle is that well-tried ideas should be generalised, where the generalisation is apparently natural; this has been applied mainly in generalising ML varstructs to HOPE patterns, in broadening the structure of declarations (following Cardelli's declaration connectives which go back to Robert Milne's Thesis), and in allowing exceptions which allow escape with values of arbitrary type (generalising the original failure construct of ML).

A third principle is to specify the language completely, so that programs will port from one correct implementation to another with minimum fuss. This entails, first, precision in concrete syntax (I agree with Cardelli that abstract syntax is in some respects more important - but we do not all have structure editors yet, and humans still communicate with each other in concrete syntax!); second, it entails exact evaluation rules (e.g. we have to specify the order of evaluation of two expressions, one applied to the other, just because of the escape mechanism).

2. The bare language

2.1 It is convenient to present the language first in a bare form, containing enough on which to base the semantic description given in Section 3. Things omitted from the bare language description are:

(1) Derived syntactic forms, whose meaning derives from their equivalent forms in the bare language (Section 6).

(2) Directives for introducing infixes and type abbreviations, and specifying type environments (Section 4).

(3) Standard types (Section 5).

(4) Input and Output, based on the standard type "stream", and external ML files (Section 8).

(5) References and equality (Section 7).

(6) Type checking (Section 9).

The composite expression forms are application, type constraint, tupling, raising and handling exceptions, local declaration (using let) and function abstraction. Varstructs are (except for the wildcard $) a subclass of expressions used in value bindings. Declarations may declare value variables (using value bindings), types with associated constructors or operations (using type bindings), and exceptions; apart from this, one declaration may be local to another (using local) and sequences of declarations are allowed. An ML program is just a sequence of declarations or directives; thus – omitting the directives which have purely syntactic effect – it is just a single declaration. The bare syntax is in Section 2.8 below; first we consider lexical matters.

2.2 Reserved words

The following are the reserved words used in the complete language. They may not be used as identifiers. In this document the alphabetic reserved words are always underlined.

abstype and case do else end escape
exception fun handle if in infix left let
local longtype nonfix of op or raise rec
shorttype spec then type use val with where while

( ) [ ] { } , : ; . | #
$ & ? == -> <=>

2.3. Numerals

A numeral is any non-empty sequence of digits.

2.4. String constants

A string constant is any sequence of printable characters or spaces enclosed between quotes ("), but within which any quote symbol is preceded by the escape character \ . Use of \ in strings has meaning as follows:

\1 .. \9	One to nine spaces	\E	Escape
\0	Ten spaces	\N	Null (Ascii 0)
\R	Carriage return	\D	Del (Ascii 127)
\L	Line feed	\^c	Ascii control character c
\T	Tabulation	\c	c (any other character)
\B	Backspace		

2.5 Identifiers

Identifiers are used to stand for five different syntax classes, which - if we had a large enough character set - would be disjoint:

value variables	(var)
value constructors	(con)
type variables	(tyvar)
type constructors.	(tycon)
exception identifiers	(exid)

An identifier is either alphanumeric: any sequence of letters, digits, primes (') and underbars (_) starting with a letter or prime, or symbolic: any sequence of the following symbols

! # & + - / : < = > ? @ \ ^ ~ | *

In either case, however, reserved words are excluded.

A type variable (tyvar) may be any alphanumeric identifier starting with a prime. The other four classes (var, con, tycon, exid) are represented by identifiers not starting with a prime. Thus type variables are disjoint from the other four classes. Otherwise, the syntax class of an occurrence of identifier id is determined thus

(1) In types, id is a type constructor, and must be within the scope of the type binding which introduced it.

(2) Following exception, raise or handle id is an exception identifier.

(3) Elsewhere, id is a value constructor if it occurs in the scope of a type binding which introduced it as such, otherwise it is a value variable.

It follows from (3) that no value binding can make a hole in the scope of a value constructor by introducing the same identifier as a variable, since this identifier must stand for the constructor in any varstruct.

The syntax-classes var, con, tycon and exid all depend on which bindings are in force, but only the classes var and con are necessarily disjoint. The context determines (as described above) to which class each identifier occurrence belongs.

An identifier may be given infix status by the <u>infix</u> command; This status only pertains to its use as a var or a con. If id has infix status, then "exp1 id exp2" (resp. "vs1 id vs2") may occur wherever the application "id (exp1, exp2)" (resp "id(vs1, vs2)") would otherwise occur. On the other hand, non-infixed occurrences of id must be prefixed by the keyword "<u>op</u>". Infix status is cancelled by the <u>nonfix</u> command. On ML files only standard infixes (e.g. "+", "::") are assumed; after reading the file, the previous infix or nonfix status of every identifier is resumed (but see 4.1 for a refinement of this rule).

2.6 Comments

A comment is any character sequence within curly brackets { } in which curly brackets are properly nested.

2.7. Lexical analysis

Each item of lexical analysis is either a reserved word or a numeral or a string constant or an identifier; comments and non-visible characters separate items and are otherwise ignored (except spaces within string constants). At each stage the longest next item is taken.

Note: As a consequence of this simple approach, spaces are needed sometimes to separate identifiers and reserved words. Two examples are

a := !b not a:=!b (assigning contents of b to a)
~ : int→int not ~:int→int (unary minus qualified by its type)

Rules which allow omission of spaces in such examples, such as adopted by Cardelli in VAX ML, also forbid certain symbol sequences as identifiers and – more importantly – are hard to remember; it seems better to keep a simple scheme and tolerate a few extra spaces.

2.8 The bare syntax

Conventions : (1) {…} means optional. (2) For any syntax class s, s_seq ::= s | (s1,…,sn). (3) Alternatives are in order of decreasing precedence, and L(R) means left(right) association. (4) Parentheses may enclose any named syntax class

EXPRESSIONS exp

aexp ::=

var	(variable)
con	(constructor)
(exp)	

exp ::=

aexp	
exp aexp	(application)
exp : ty	(constraint)
exp1,…,expn	(tuple)
raise exid exp	(raise exception)
let dec in exp end	(local decl^n)
exp handle exid match	L (handle exception)
fun match	(function)

match ::=
vs1.exp1 | ⋯ | vsn.expn

DECLARATIONS dec

dec ::=

val vb	(values)
type tb	(types)
abstype tb with dec end	(abstract types)
exception id1{:ty1} and ⋯ and idn{:tyn}	(exceptions)
local dec1 in dec2 end	(local decl^n)
dec1 ; dec2	R (sequence)

COMMANDS com

com ::=

dec ;	(declaration command)
directive ;	(directive command)

VARSTRUCTS vs

avs ::=

$	(wildcard)
var	(variable)
con	(constant) constructor ?
(vs)	

vs ::=

avs	
con avs	(construction)
vs : ty	(constraint)
vs1 , … , vsn	(tuple)

VALUE BINDINGS vb

vb ::= redundant

vs {:ty} == exp	(simple)
vb1 and vb2	(simultaneous)
rec vb	(recursive)

TYPE BINDINGS tb

tb ::=

{tyvar_seq} tycon == constrs	(simple)
tb1 and tb2	(simultaneous)
rec tb	(recursive)

constrs ::=

con1 {of ty1} | … | conn {of tyn}	(constructors)

TYPES ty

ty ::=

tyvar	(type variable)
{ty_seq} tycon	(type construction)
ty1 # ⋯ # tyn	(tuple type)
ty1 → ty2	R (function type)

A PROGRAM IS A SERIES OF COMMANDS

3. Evaluation

3.1 Environments and Values

Evaluation of phrases takes place in the presence of an ENVIRONMENT and a STORE. An ENVIRONMENT E has two components: a value environment VE, associating values to variables and to value constructors, and an exception environment EE associating exceptions to exception identifiers. A STORE S associates values to references, which are themselves values. (A third component of an environment, a type environment TE, is ignored here since it is relevant only to type checking and compilation, not to evaluation.)

An exception e, associated to an exception identifier exid in any EE, is an object from which exid may be recovered. A packet $p=(e,v)$ is an exception paired with a value. Packets are not values. Besides possibly changing S (by assignment), evaluation of a phrase returns a result as follows:

obscure

Phrase	Result
Expression	v or p
Declaration	E or p
Value binding	VE or p
Type binding	VE or p

For every phrase except a handle expression, whenever its evaluation demands the evaluation of an immediate subphrase which returns a packet p as result, p is also the result of the phrase.

A function value f is a partial function which, given a value, may return a value or a packet; it may also change the store as a side-effect. Every other value is either a constant (a nullary constructor), a construction (a constructor with a value), a tuple or a reference.

3.2 Environment manipulation

We may write $\langle(var1, v1) \ldots (varn, vn)\rangle$ for a value environment, where the vari are distinct. Then $\langle\rangle$ is the empty VE, and

not commutative!

VE1 + VE2 means the VE in which the associations of VE2 supersede those of VE1. Similarly for exception environments. If $E = (VE, EE)$ and $E' = (VE', EE')$, then $E + E'$ means $(VE+VE', EE+EE')$, $E + VE'$ means $E + (VE', \langle\rangle)$, etc. This implies that an identifier may be associated both in VE and in EE without conflict.

bound

better to write VE1 ; VE2 ?

3.3 Matching varstructs

The matching of a varstruct vs to a value v either *fails* or yields a VE. Failure is distinct from returning a packet, but will result in this when all varstructs fail in applying a match to a value (see 3.4). In the following rules, if any component varstruct fails to match then the whole varstruct fails to match.

The following is the effect of matching cases of vs to v:

\$: the empty VE is returned.

var : the VE $\langle(var, v)\rangle$ is returned

con{vs}: if v = con{v'}, then vs is matched to v'; else failure.

$vs1, \ldots, vsn$: if $v = (v1', \ldots, vn)$ then vsi is matched to vi returning VEi, for each i; then VE1+…+VEn is returned.

vs : ty : vs is matched to v.

[What about requirement that varstructs be linear, so that VEi are all over disjoint sets of ids].

3.4 Applying a match

[Simple left to right match with failure exception]

Assume environment E. Applying match $m = vs1.exp1 | \ldots | vsn.expn$ to value v returns a value or packet as follows:

Each vsi is matched to v in turn, from left to right, until one succeeds returning VEi; then $expi$ is evaluated in E+VEi. If none succeeds, then the packet (e_{STRING}, "FAILED TO MATCH VALUE V") is returned, where e_{STRING} is the standard exception always associated in E with the standard exception identifier STRING of type string, and V is the print form of value v.

Thus, for each E, a match m denotes a function value.

3.5 Evaluation of expressions

Assume environment $E = (VE, EE)$. Evaluating an expression exp returns a value or packet as follows, by cases of exp:

var : returns value $VE(var)$

con : returns value $VE(con)$

exp aexp : exp is evaluated, returning function value f or p; then aexp is evaluated, returning value v or p; then $f(v)$ is returned.

exp1, …, expn: the expi are evaluated in sequence, from left to right, returning vi respectively; then $(v1, \ldots, vn)$ is returned.

raise exid exp : exp is evaluated, returning value v; then packet (e, v) is returned, where $e = EE(exid)$.

exp handle exid match : exp is evaluated; if exp returns v then v is returned; if exp returns $p = (e, v)$ then (1) if $e = EE(exid)$ then match is applied to v, (2) if $e \neq EE(exid)$ then p is returned.

let dec in exp end : dec is evaluated, returning E'; then exp is evaluated in $E + E'$.

fun match : f is returned, where f is the function of v gained by applying match to v in environment E.

exp : ty : exp is evaluated.

3.6 Evaluation of value bindings

Assume environment $E = (VE, EE)$. Evaluating a value binding vb returns a value environment VE' or a packet as follows, by cases of vb:

$vs\{: ty\} == exp$: exp is evaluated in E, returning value v; then vs is matched to v; if this returns VE', then VE' is returned, and if it fails then the packet $(e_{STRING}$, "FAILED TO BIND VALUE V") is returned, where e_{STRING} and V are as described in 3.4.

$vb1$ and $vb2$: $vb1$ is evaluated in E, returning $VE1$; then $vb2$ is evaluated in E, returning $VE2$; then $VE1 + VE2$ is returned.

rec vb : vb is evaluated in E', returning VE', where $E' = (VE + VE', EE)$. Because the values bound by evaluating vb must be function values, E' is well defined by "tying knots" (Landin).

3.7 Evaluation of type bindings

The components VE and EE of the current environment do not affect the evaluation of type bindings (TE affects their type checking and compilation). Evaluating a type binding tb returns a value environment VE′ (it cannot return a packet) as follows, by cases of tb:

{tyvar_seq} tycon == con1 {of ty1} | ... | conn {of tyn} : The value environment VE′ = ⟨(con1, v1), ..., (conn, vn)⟩ is returned, where vi is either the constant value coni (if "of tyi" is absent) or else the function value $v \mapsto$ coni v. Note that all other effect of this type binding is handled by the compiler and type checker, not by evaluation.

tb1 and tb2 : tb1 and tb2 are evaluated, returning VE1 and VE2 respectively; then VE′ = VE1 + VE2 is returned.

rec tb : tb is evaluated. Note again that the recursion is handled by type checking only.

3.8 Evaluation of declarations

Assume environment $E = (VE, EE)$. Evaluating a declaration dec returns an environment E' or a packet as follows, by cases of dec:

<u>val</u> vb : vb is evaluated, returning VE'; then
$E' = (VE', \langle\rangle)$ is returned.

<u>type</u> tb : tb is evaluated, returning VE'; then
$E' = (VE', \langle\rangle)$ is returned.

<u>abstype</u> tb <u>with</u> dec <u>end</u> : tb is evaluated, returning VE'; then dec is evaluated in $E + VE'$, returning E'; then E' is returned.

<u>exception</u> exid{: ty} : a new exception e is generated (from which the exception identifier exid may be recovered), and
$E' = (\langle\rangle, \langle(exid, e)\rangle)$ is returned.

<u>local</u> dec1 <u>in</u> dec2 <u>end</u> : dec1 is evaluated, returning $E1$; then dec2 is evaluated in $E + E1$, returning $E2$; then $E' = E2$ is returned.

dec1 ; dec2 : dec1 is evaluated, returning $E1$; then dec2 is evaluated in $E + E1$, returning $E2$; then $E' = E1 + E2$ is returned.

Note that each declaration is defined to return only the new environment which it makes, but the effect of declarations composed by ";" is to accumulate environments.

4. Directives

There are three kinds of directive; they are for establishing infix status of value variables and constructors, for introducing and cancelling type abbreviations, and for specifying an assumed type environment (mainly in external programs, which allows their precompilation).

4.1 Infix directives

<u>infix</u> id1 ... idn
<u>nonfix</u> id1 ... idn

The <u>infix</u> directive introduces infix status for each idi (as a value variable or constructor), and the <u>nonfix</u> directive cancels it. While id has infix status, each occurrence of it must be infixed or else preceded by <u>op</u>. Several standard functions and constructors have infix status (see Appendix 5) with precedence and left or right association; user-defined infixes are all left associative with precedence 0 (this is to simplify the <u>spec</u> directives – see 4.3 below).

An external program assumes initially only standard infix statuses. After it is imported (by the <u>use</u> declaration) previous infix or nonfix statuses are resumed, except that any id declared with uncancelled infix status within the external program, whose declaration <u>outstays</u> the external program, retains its infix status. This is naturally achieved by recording infix status in the type environment TE.

?

4.2 Type abbreviation directives

<u>shorttype</u> {tyvar_seq1}tycon1 == ty1 <u>and</u> ... <u>and</u> {tyvar_seqn}tyconn == tyn
<u>longtype</u> tycon1 ... tyconn

The shorttype directive has no semantic significance; it merely allows any instance (by substitution for type variables) of tyi to be replaced – both in programs and on output – by the corresponding instance of {tyvar-seqi}tyconi. In abbreviating types on output, most recent shorttype directives are matched first, and each type is matched before its subtypes.

The longtype directive cancels the shorttype status of tyconi, which reverts to its previous rôle (if any) as a type constructor.

There are no standard type abbreviations. An external program initially assumes no type abbreviations, and any which it introduces are later forgotten.

4.3. Specification directives

The purpose of specification directives is to specify type constructors, value variables and constructors (with their infix status), and exception identifiers, which are used but not declared in a program. This is mainly to make external programs syntactically self-contained and to allow their precompilation. The directives would normally occur at the head of these programs, but are only necessary if precompilation is done — and then need only occur before the specified items are mentioned.

spec val {op}id1:ty1 and ... and {op}idn:tyn
spec type {tyvar_seq1}tycon1 and ... and {tyvar_seqn}tyconn
spec type tb
spec exception id1:ty1 and ... and idn:tyn

Note that type constraints are required, not optional as in declarations. The spec val directive specifies the generic types of value variables which are to be assumed (and infix status, if any). When the program is loaded — precompiled or not — by use, the current type environment TE must record each idi as a variable (or value constructor) having type tyi' of which tyi is an instance, and infix status iff specified. The first form of spec type directive indicates type constructors which must be in force (declared by type or abstype) on loading. The second form indicates types-with-constructors which must be in force on loading. This form is needed for precompilation of a program which uses the specified constructors in varstructs, or uses equality (=) on the specified types; the full form (using a tb) is adopted because precompilation must be able to check the conditions to be satisfied by the varstructs in a match: see Section 10 under (2).

The spec exception directive specifies exception identifiers which must be in force on loading.

As an example, if a program assumes the existence of a type 'a TREE, with three constructors (one infixed), it may specify

spec type 'a TREE == nulltree | tip of 'a | op constree of 'a TREE # 'a TREE ;

and then the type environment TE on loading must record TREE as a unary type constructor with these three value constructors at the same (or more general) types. But if the program does not use the constructors in varstructs, nor uses equality on trees, then it may merely specify

spec type 'a TREE ;
spec val nulltree : 'a TREE and tip : 'a -> 'a TREE
and op constree : 'b TREE # 'b TREE -> 'b TREE ;

In this case, the loading environment may be as above, or may merely record the unary type constructor TREE and three operations (constructors or not) with appropriate type.

5. Standard type constructors

The bare language provides the function-type constructor, →, and for each $n \geq 2$ a tuple-type constructor $\#_n$. $(ty1, \ldots, tyn)\#_n$ is written ty1#…#tyn. Beside these, the following are standard:

Type constants (nullary constructors): unit, bool, int, string, stream.
Unary type constructors: list, ref

The constructors unit, bool and list are fully defined by the following assumed declarations (note that "::" is a standard infix):

<u>type</u> unit == () <u>and</u> bool == true | false {"()" is a special constant}
<u>and</u> <u>rec</u> 'a list == nil | <u>op</u> :: <u>of</u> 'a # 'a list ;

The type constant int (integers) is equipped with constants 0, 1, ~1, 2, ~2, …,
The type constant string is equipped with constants as described in 2.4.
The type constant stream is for Input/Output; see Section 8.
The type constructor ref is for constructing reference types; see Section 7.

The standard functions over all these types are listed in Appendix 5. There are not a lavish number of them; we envisage libraries of functions provided by each implementation, together with their ML declarations (though they may be implemented more efficiently).

6. Standard Derived Forms

6.1 Expressions and Varstructs

	DERIVED FORM	EQUIVALENT FORM
EXPRESSIONS	escape exp	raise STRING exp
	exp trap match	exp handle STRING match
	exp1 ? exp2	exp1 trap ($. exp2)
	case exp of match	(fun match) exp ← Is this like let for typechecking [yes!]
	if exp then exp1 else exp2	case exp of (true. exp1 | false. exp2)
	exp1 or exp2	if exp1 then true else exp2
	exp1 & exp2	if exp1 then exp2 else false
	exp where dec end	let dec in exp end
	exp1 ; exp2	let val $ == exp1 in exp2 end
	while exp1 do exp2	let val rec f == fun(). if exp1 then (exp2; f()) else () in f() end
	[exp1; ... ; expn]	exp1 :: ... :: expn :: nil
VARSTRUCTS	[vs1; ... ; vsn]	vs1 :: ... :: vsn :: nil

The derived form may be implemented more efficiently than its equivalent form. The type-checking of each derived form is defined by that of its equivalent form, except in the case expression which is treated more like the let expression (see Section 9).

The binding power of all bare and derived forms is shown in Appendix 1. Important: ";" has weakest binding power in both expressions and declarations.

The escape and trap forms refer to a predefined exception identifier "STRING" of type "string"; This models exactly the old ML failure forms.

6.2 Bindings and declarations

	DERIVED FORM	EQUIVALENT FORM
VALUE BINDINGS	var avs1 ··· avsn {:ty} == exp	var == fun avs1. ···. fun avsn. exp {:ty}
	var avs1 {:ty} == exp1 | ··· | var avsn {:ty} == expn	var == fun avs1. exp1 {:ty} | ··· | avsn. expn {:ty}
TYPE BINDINGS	{tyvar_seq} tycon <=> ty	{tyvar_seq} tycon == mk_tycon of ty
DECLARATIONS	exp	val it == exp

Notes: The first derived value binding allows Curried function definitions; the second allows separate equations in defining non-Curried functions with several patterns. Separate equations for Curried function definitions are forbidden.

The derived type binding is for isomorphic types, introducing an explicit "abstraction" constructor consisting of "mk_" prefixed to the type constructor.

The derived declaration is principally for treating expressions at top-level as degenerate declarations. The variable "it" is just a normal variable, but is useful for referring to the value returned by the last top-level expression.

7. References and equality

7.1 References and assignment

Following Cardelli, references are provided by the type constructor "ref". Since we are sticking to monomorphic references, there are two overloaded functions available at all monotypes μ:

(1) ref : $\mu \to \mu$ ref, which associates (in the store) a new reference with its argument value. In varstructs, ref : 'a $\to$ 'a ref may be used polymorphically.

(2) <u>op</u> := : μ ref # μ $\to$ unit, which associates its second (value) argument with its first (reference) argument in the store, and returns ( ) as result.)

The polymorphic contents function "!" is provided, but may be derived as follows: "<u>val</u> ! (ref x) == x".

7.2 Equality

The overloaded equality function <u>op</u> = : γ # γ $\to$ bool is available at all types γ built from references by tuple type and type constructors declared by <u>type</u> (including "list"), not by <u>abstype</u>:

& bool (& int)

$$\gamma ::= \text{ty ref} \mid \gamma_1 \# \cdots \# \gamma_n \mid \{\gamma\}\text{tycon}$$

On references, equality means identity. On objects of other types γ it is recursively defined on the structure of γ in the natural way. The inequality function <u>op</u> <> is also provided.

8. Input and Output

8.1 Input/output in programs

Following Cardelli's suggestion, Input/Output is done using by streams (of characters). Filenames are strings. A stream is created from a file by the function "getstream : string → stream", and a file is created (or re-created) from a stream by the function "putstream : string → stream → unit". These are the only operations which refer to files. Getstream doesn't change the file (and treats nonexistent files as empty); putstream doesn't change the stream.

"stream" may be regarded as a pre-declared abstract type; it provides as operations the two functions getstream and putstream, and four stream operations:

- newstream : unit → stream (Creates an empty stream)
- nextstring : stream → int → string (Returns a string of given length from the front of a stream, leaving the stream unchanged)
- instring : stream → int → string (Removes a string of given length from the front of a stream).
- outstring : stream → string → unit (Appends a string to a stream).

Terminal I/O is represented by the standard stream variables input, output. The variable "input" denotes the (infinite!) stream of characters from the keyboard; this stream may be terminated by e.g. "CTRL Z", whereupon input will again stand for the stream of characters which will follow. The variable "output" denotes the (initially empty) stream of characters sent to the screen.

The functions getstream and putstream will escape with "getstream", "putstream" in case of inadmissible file names, protection status etc (this is implementation dependent, and not reflected in the description given below). The functions nextstring, instring escape with "nextstring", "instring" if their integer argument is negative or too large. When necessary, nextstring and instring must prompt for a further segment of the input stream. Applying nextstring or instring to the output stream, or outstring to the input stream, generates an escape.

?

The following abstract type definition of stream is to be regarded as evaluated once, when a new file area is created. (The description is not an implementation suggestion! It is only for clarification.) It establishes a directory: a map from strings (file names) to string references (representing files). All uses of getstream and putstream by ML refer to the directory, and other file-handling operations outside ML are thought of as working through these two functions. To shorten the description, we assume a function "split : string # int → string # string" such that split(s, n) = (s1, s2) where s1^s2 = s and s1 contains n characters; split escapes if n is negative or too large.

```
abstype stream <=> string ref with
        {first initialise the file directory with all files empty}
  val directory : string -> string ref == fun s. ref "";
        {next define the file operations; error conditions are not represented here}
  val getstream (filename : string) : stream == mk_stream(ref(!(directory filename)))
  and putstream (filename : string)(mk_stream(ref s) : stream) : unit ==
            directory filename := s ;
        {finally define the stream operations} ;
  val newstream () : stream == mkstream (ref "")
  and nextstring (mk_stream r : stream) (n : int) : string ==
      let val s, $ == split (!r, n) in s end ? escape "nextstring"
  and instring (mk_stream r : stream) (n : int) : string ==
      let val s, s' == split ( !r, n) in r := s' ; s end ? escape "instring"
  and outstring (mk_stream r : stream) (s : string) : unit == r := !r ^s
  end {of stream} ;
```

Note that "nextstring s 1" escapes if s is empty, allowing a test for the empty stream to be defined. The function nextstring is needed also for "peeking" at streams without changing them.

8.2 External programs

An ML program on a file may be evaluated using the use declaration; since a program (ignoring directives) is just a declaration, the meaning of this is well-defined. Note that (Section 10 (7)) a use declaration may not be inside a match. This means it cannot be evaluated more than once. It may, however, be in a top-level context such as

local use "MYPROG" in dec end ;

An external program assumes only standard infix statuses, and standard (i.e. no !) type abbreviations; if it introduces any of these, they last only for the program itself. Thus the effect of directives in the main program is not disturbed by a use declaration†. An external program may contain further use declarations.

By use of spec directives, an external program may specify bindings which it requires in any environment to which it is loaded. This allows external programs to be precompiled. The use declaration may refer both to source and to precompiled external programs.

† There is one exception: infixed functions declared in the external program retain their infix status (see 4.1).

9. Type-checking

The type-checking discipline is exactly as in original ML, and therefore need only be described with respect to new phrases.

In a match m = vs1.exp1 | ... | vsn.expn , the types of all vsi must be the same (ty), and if variable var occurs in vsi then all free occurrences of var must have the same type as its occurrence in vsi. There is one relaxation of this rule — the case expression; see below. In addition, the types of all the expi must be the same (ty′). Then ty → ty′ is the type of m.

The type of "fun match" is the type of the match. The type of "exp handle exid match" is ty′, where exp has type ty′, match has type ty → ty′, and exid has type ty. The type of "raise exid exp" is arbitrary, but exp and exid must have the same type. Thus the type of an exception may be polymorphic; exid is only required to have the same type at all occurrences within the scope of its declaration (and this must be an instance of any type qualifying the declaration).

The type of "case exp of match" is ty′, where exp has type ty and match has type ty → ty′. However in the match, for each var occurring in a vsi, each occurrence of var in expi is only required to have as type a generic instance of its type in vsi; in this respect, case is similar to let.

A type variable is only explicitly bound (in the sense of variable-binding in λ-calculus) by its occurrence in tyvar_seq in the type binding "{tyvar_seq} tycon == constrs", and then its scope is "constrs". This means that bound uses of 'a in both tb1 and tb2 in the type binding "tb1 and tb2" bear no relation to each other.

Otherwise, repeated occurrences of a (free) type variable may serve to link explicit type constraints. The scope of such a type variable is the smallest top-level command in which it occurs.

The type-checker refers to the type environment (TE) component of the environment, and records its findings there. Details of TE are not given in this report; they are compatible with what is done in current ML implementations, except that value constructors (and their types) are associated with the type constructors to which they belong.

10. Syntactic restrictions

(1) No varstruct may contain two occurrences of the same variable.

(2) In a match "vs1.exp1|...|vsn.expn", the varstruct sequence vs1,..,vsn should be non decreasing and exhaustive. That is, for i<j vsi must not match all the values which vsj matches, and every value (of the right type) must be matched by some vsi. The compiler must report a violation of this restriction, but should still compile the match. The restriction applies to all derived forms; in particular, this means that in the Curried function binding var avs1...avsn{:ty}==exp, each separate avsi should be exhaustive by itself.

(3) For each value binding "vs{:ty}==exp" the compiler must issue a report (but still compile) if either vs is not exhaustive or vs contains no variable. This will (on both counts) detect errors like "val nil == exp" in which the user expects to declare a new variable nil (whereas the language dictates that nil is here a constant varstruct, so no variable gets declared). Cardelli points out this danger. However, these warnings should not be given when the binding is a component of a top-level declaration; e.g. "val x::l == exp" is not faulted by the compiler at top level, but may of course generate an escape "FAILED TO MATCH ..." (see Section 3.4).

(5) In every instance of "{tyvar_seq}tycon" the tyvar_seq must contain no type variable more than once. The right hand side of a simple type binding may contain only the type variables mentioned on the left.

(4) For each value binding "vs{:ty}==exp" within rec, and for each derived form of such a binding, exp must be of the form "fun match".

(6) In "let dec in exp end" and "local dec in dec′ end" no type constructor exported by dec may occur in the type of exp or in the type of any variable or value constructor exported by dec′. The "where" form inherits a similar restriction.

(7) The declaration form use "filename" must not occur within a match, nor within a derived form if it would occur within a match in the equivalent form.

(8) Every top-level exception declaration must be explicitly constrained by a monotype.

11. Record of discussions

In this section I try to record some of the views which people expressed about my tentative proposal and alternatives to it. There were two very helpful group discussions, with eight or ten people present, and several smaller meetings at INRIA, Cambridge and Edinburgh. Dave MacQueen was most active in these, gathering views together as much as possible. A lot of progress was made; even so, I sensed that more group discussions would have wasted people's time in proportion to their benefit; the best thing was to collect views in smaller discussions and try to find a coherent proposal which gained from them, while avoiding weak compromises.

More things were discussed than are reported below, but I hope to have recorded the more important points of contention.

11.1 Abstract types and modules

In original ML – and in my Tentative proposal – the type isomorphism declaration "abstype {tyvar_seq} tycon ⇔ ty with ..." was a basic form. In this proposal, thanks to Rod Burstall and others, it becomes a derived form which is a special case of something more powerful.

The present form seems to represent the natural effect of introducing HOPE data types into a consolidated ML, without further language development. Note that the data-types-with-constructors idea is itself a natural rounding off, generalising the fixed varstructs of old ML to allow user-defined varstructs.

On the other hand, the strongest pressure for developing the language was in the case of modules. The argument – from Cardelli, Mitchell, Paulson (at least) – is certainly strong. First, modules allow separate compilation; secondly, they allow specification to be separated from implementation. In richer versions, they can be parametric, and can even be values of a new kind.

This pressure was balanced by others who favoured the conservative course of consolidation.

There are virtues and dangers in both courses of action. Having committed myself to propose something definite (or at least to monitor such a proposal) I am acutely aware of the need to act quickly – otherwise impetus is lost, people's viewpoints change and the chances of a coherent (if not perfect) design recede dramatically! So it seemed better to leave modules out of <u>this</u> proposal because

(1) The original aim was consolidation

(2) It isn't clear to me – even after reading a nice intentionally modest module proposal from Luca Cardelli – where to draw the line in what modules provide.

(3) A proposal for a language with modules should come from those who have experience of them, which I do not.

However, the practical need for separate compilation doesn't demand modules. To satisfy this need, the <u>spec</u> directives (Section 4.3) seem rather simple and can be seen as consolidation, not extension. Note that they do not change the meaning of programs at all; the unifying idea, that a program is (ignoring the directives) just a single declaration, is preserved.

This does not prevent a version of ML with modules being adopted, say within the next year or two, as a development of the present proposal.

11.2 Escapes and Traps

The exception mechanism in this proposal came from Alan Mycroft; it also owes something to an idea from Brian Monahan. The views expressed on escaping and trapping ranged from contentment with the existing ML mechanism, in which escapes can only carry tokens as values, to approval of the present proposal in which exceptions can carry arbitrary values, and can even be polymorphic. (To be fair, there was also the view that both escapes and references are impure and should be banned; this was a minority view).

The proposal is easy to implement, easy to describe semantically, and specialises smoothly to the existing mechanism. Thus, although none could predict its effect on programming style, it seems safe to adopt it. No other scheme was put forward which fits so well with the polymorphic type discipline.

11.3 Data types and parameter matching

The proposal to import the HOPE data constructs, with use of constructors in varstructs, met with approval from most people. It could have been combined with Cardelli's records and variants, allowing named fields in records, but to do so would have led to an embarrassment of riches. Luca Cardelli was kind enough to accept it.

There was a lot of discussion about the constraints that should be placed upon the varstructs in a match, and the order in which these should be matched. The main point is that, if the set of varstructs in a match is required to be closed under unification, and the order of matching is from more specific to less specific (which still leaves freedom) the implementation can be very efficient. On the other hand, left-to-right order is easy for users to understand.

It came out in discussion with Dave MacQueen (who has studied this question in detail) that in fact the above requirement is not essential to gain the efficient implementation. Roughly, it appears that a compiler can process an arbitrary match to attain the unification-closed property, and can then adopt an efficient order of matching which is semantically equivalent to left-to-right. I have therefore proposed left-to-right order, to gain a simple universal rule in the language, and the compiler will issue a warning if a more specific varstruct follows a less specific one in a match, or if the varstructs in a match are not exhaustive.

11.4 Input/Output

There was not much general discussion of I/O, but Luca Cardelli proposed the use of streams (in place of my tentative proposal) and this seems very clean.

The proposed standard functions are obviously too few for convenient use. But library functions should be defined from them. Cardelli proposed a function "copystream : stream → stream". I left this out because Kevin Mitchell suggested that it might demand a special choice of implementation. In any case, it can be defined if we are prepared to use an auxiliary file COPY:

val copystream (s) == (putstream "COPY" s ; getstream "COPY") ;

In my tentative proposal I suggested overloaded functions "read" and "write", which would read and write data of arbitrary (mono)type, in the form of ML constructions. This seems a bad idea; it would involve delicate interaction with the ML compiler (perhaps not only the parser) to take account of the currently defined value constructors.

11.5 Clausal forms for function declaration

There was some discussion of how rich a form of clausal function declaration to allow. The present proposal goes as far as allowing e.g.

```
val rec member (x, nil) == false
      | member (x, y::l) == x=y or member(x,l)
```

but forbidding the Curried form

```
val rec member x nil == false
      | member x (y::l) == x=y or member x l
```

It was agreed that the second form requires careful explanation of when pattern matching of arguments fails. It was not fully agreed whether the form is important enough to justify the explanation. With this doubt, the decision to forbid the Curried form seems justified, acknowledging that to mix Currying with alternative patterns risks confusion.

11.6 Tuples vss pairs

Most people with whom I discussed it prefer the tuple type ty1#...#tyn to the simple pair type ty # ty'. This is the scheme adopted in HOPE. I now prefer it too. Thus ty1#...#tyn abbreviates (ty1,...,tyn)$\#_n$, where $\#_n$ is a different type constructor for each $n \geq 2$. It follows that (x,y,z) is different from (x,(y,z)), unlike original ML. I guess that the choice of a simple pair type in original ML dates from the time when we had only a small finite collection of standard type constructors (→ # +) and no user defined ones.

11.7 Syntactic matters

(1) end : Everyone has different dos and donts (or ods and tnods) about terminating keywords. Old ML gave people headaches with things like where. There is no perfect compromise between ambiguity and verbosity. In the absence of inspiration, at least the present proposal can probably be remembered:

end delimits only let, where, local and abstype

MNEMONIC: end is needed in any construct which sticks a declaration to other things with alphabetic keywords!

See also (10) below

(2) Atomic varstructs: This wasn't discussed, but has certainly caused trouble in the past. We don't want

val f x:int, y:bool == ...

in place of "val f (x:int, y:bool) == ". In fact, we want

val f (x:int, y:bool) : string == ...

to declare a function of type int # bool → string. This points to having argument varstructs atomic (closed) in these sugared forms of declaration. On the other hand, we do wish to allow non atomic varstructs in function abstraction, e.g.

fun x:int, y:bool . --- , or fun x, y. ---

To avoid possible confusion, it seems wise to omit Curried fun forms like

fun (x:int) (y:bool). ...

because the varstructs here should be atomic, and this would blur the simple mnemonic that parameter varstructs need only be atomic in derived function declaration forms. Besides, Curried fun forms are not too frequent in most practical programming styles.

(3) Distfixes: again, there wasn't much general discussion of the need for these. Don Sannella gave helpful advice about how to manage them. In the end, I couldn't find enough enthusiasm in anyone (particularly in myself) to justify the careful explanation that they need – surprisingly tricky when you try it – so they are omitted.

(4) Abstract vss concrete syntax: Luca Cardelli points out that eventually (soon for some people) structure editors will reduce or remove our worries about precedence, delimiters, lexical analysis etc. Hence we should be sure to define the abstract syntax of the language, and only recommend the concrete syntax. It is a relief to adopt this view, particularly as human beings degenerate into animals when they argue about concrete syntax, just as they do when they drive motor cars! But the recommendation needs to be quite strong, because people communicate with other people in concrete syntax. This proposal hasn't dismissed abstract syntax; it can be deduced quite easily from the recommended concrete forms.

(5) Type abbreviations: There was a preponderance (among those who have written fair-sized programs) of the need for type abbreviation. The consensus – not that everyone expressed a view – seems to be that it is a dirty but expedient device, and that it should not have semantic significance.

(6) let: There was some doubt as to whether let should be (a) on a par with type as a way of characterising a binding, or (b) an indication that a binding is of limited scope. Possibly this doubt arose from the unfortunate way in which old ML confused the two, which people had come to accept. The majority favoured nonconfusion, if it could be achieved without pain. The present val (on a par with type) was accepted by at least some people – certainly as preferable to var which was used in this rôle in my Tentative proposal.

(7) Underbar: most people want it in identifiers. Some people are happy to have it as a wildcard varstruct as well. I don't like this; the two uses can lead to some odd varstructs, e.g. "_a_b_", which in this proposal is written "$ a_b $", meaning the same as "op a_b($,$)".

(8) "=" and "==": No one liked my use of "⇐" in bindings, in my Tentative proposal, to avoid overloading the symbol "=". Luca Cardelli is still happy to overload "=", but a fair compromise seems to be "==" in bindings.

(9) Sequence connectives: there are many different places where sequences of like forms occur (identifiers in some directives, bindings in declarations, tuples). The temptation is to use a comma as the connective rather often. The proposal avoids this; comma is only used for tupling – including tuples of type arguments of type constructors. "and" is used for bindings in declarations and for the degenerate bindings which occur in spec directives. When a sequence of bare identifiers is required, a space is the connective.

One connective, the semi-colon, is a bit overloaded; this is tolerated in deference to the ML tradition of explicit list formation and the general programming tradition of statement (or command) sequencing.

(10) Delimiters again: The low precedence of semicolon will avoid some tendencies to error in syntactic grouping. Beyond this, the habit of always enclosing a match in parentheses will remove other such tendencies. This habit could even be made a requirement; on balance I think I prefer to leave it optional.

APPENDIX 1 : EXPRESSIONS

aexp ::=

{op} var	(variable)
{op} con	(constructor)
[exp1; ··· ; expn]	(list)
(exp)	

exp ::=

aexp	
exp aexp	(application)
exp : ty	(constraint)
exp1 id exp2	(infixed application)
exp1 & exp2	(conjunction)
exp1 <u>or</u> exp2	(disjunction)
exp1, ···, expn	(tuple)
<u>raise</u> exid exp	(raise exception)
<u>escape</u> exp	(escape)
<u>if</u> exp <u>then</u> exp1 <u>else</u> exp2	(conditional)
<u>while</u> exp1 <u>do</u> exp2	(iteration)
<u>let</u> dec <u>in</u> exp <u>end</u>	(local declaration)
exp <u>where</u> dec <u>end</u>	(post-declaration)
<u>case</u> exp <u>of</u> match	(case expression)
† exp <u>handle</u> exid match	(handle exception)
exp <u>trap</u> match	(trap)
exp1 ? exp2	(trap string)
<u>fun</u> match	(function)
exp1 ; exp2	(sequence)

match ::= vs1.exp1 | ··· | vsn.expn

† These three forms are of equal precedence and left associative

APPENDIX 2 : VARSTRUCTS and VARIABLE BINDINGS

avs ::=

$	(Wildcard)
{op} var	(variable)
con	(constant)
[vs1; ···; vsn]	(list)
(vs)	

vs ::=

avs	
con avs	(construction)
vs : ty	(constraint)
vs1 con vs2	(infixed construction)
vs1, ···, vsn	(tuple)

vb ::=

vs {:ty} == exp	(simple)
† {op} var avs1 {:ty} == exp1 | ··· | {op} var avsn {:ty} == expn	(clausal function)
† {op} var avs1 ··· avsn {:ty} == exp	(Curried function)
vb1 and vb2	(simultaneous)
rec vb	(recursive)

† In case "op var" is admissible (because var has infix status) then "avs var avs′ " may replace " op var (avs, avs′)" in these forms.

APPENDIX 3 : TYPES, TYPE BINDINGS and DECLARATIONS

ty ::=
- tyvar (type variable)
- {ty_seq}tycon (type construction)
- ty1 # ... # tyn (tuple type)
- ty1 -> ty2 (function type)

tb ::=
- {tyvar_seq}tycon == con1 {of ty1} | ... | conn {of tyn} (simple)
- {tyvar_seq}tycon <=> ty (isomorphism)
- tb1 and tb2 (simultaneous)
- rec tb (recursive)

dec ::=
- val vb (value declaration)
- type tb (type declaration)
- abstype tb with dec end (abstract type declaration)
- exception exid1 {:ty1} and ... and exidn {:tyn} (exception declaration)
- local dec1 in dec2 end (local declaration)
- use "filename" (use file as declaration)
- exp (declaration of "it")
- dec1 ; dec2 (declaration sequence)

APPENDIX 4 : DIRECTIVES, COMMANDS and PROGRAMS

directive ::=

infix id1 ... idn	(give infix status)
nonfix id1 ... idn	(cancel infix status)
shorttype {tyvar_seq1} tycon1 == ty1 and ... and {tyvar_seqn} tyconn == tyn	(type abbreviation)
longtype tycon1 ... tyconn	(cancel type abbreviation)
spec val {op} id1 : ty1 and ... and {op} idn : tyn	(specify value variables)
spec type tb	(specify types with value constructors)
spec type {tyvarseq1} tycon1 and ... and {tyvar_seqn} tyconn	(specify type constructors)
spec exception id1 : ty1 and ... and idn : tyn	(specify exception identifiers)

command ::=

dec ;

directive ;

A PROGRAM is a sequence of commands.

APPENDIX 5 : PREDECLARED VARIABLES AND CONSTANTS

NONFIXES

† nil	: 'a list	(Lists)
hd	: 'a list → 'a	(Lists)
tl	: 'a list → 'a list	(Lists)
map	: ('a → 'b) → 'a list → 'b list	(Lists)
rev	: 'a list → 'a list	(Lists)
† true	: bool	(booleans)
† false	: bool	(booleans)
not	: bool → bool	(booleans)
intofstring	: string → int	(strings)
stringofint	: int → string	(strings)
size	: string → int	(strings)
explode	: string → string list	(strings)
implode	: string list → string	(strings)
~	: int → int	(minus)
!	: 'a ref → 'a	(contents)
† ref	: $\mu \to \mu$ ref	(new reference)
getstream	: string → stream	Input/Output
putstream	: string → stream → unit	Input/Output
newstream	: unit → stream	Input/Output
nextstring	: stream → int → string	Input/Output
instring	: stream → int → string	Input/Output
outstring	: stream → string → unit	Input/Output
input	: stream	Input/Output
output	: stream	Input/Output

INFIXES (association L or R)

PRECEDENCE 5

*	: int # int → int	L	(Arithmetic)
div	: "	L	(Arithmetic)
mod	: "	L	(Arithmetic)

PRECEDENCE 4

+	: int # int → int	L	(Arithmetic)
−	: "	L	(Arithmetic)
^	: string # string → string	L	(Concatenation)

PRECEDENCE 3

† ::	: 'a # 'a list → 'a list	R	(Cons)
@	: 'a list # 'a list → 'a list	R	(Append)

PRECEDENCE 2

=	: γ # γ → bool	R	(Equality)
<>	: "	R	(Inequality)
<	: int # int → bool	R	(Integer order)
>	: "	R	(Integer order)
<=	: "	R	(Integer order)
>=	: "	R	(Integer order)

PRECEDENCE 1

o	: ('b → 'c) # ('a → 'b) → ('a → 'c)	R	(Composition)

PRECEDENCE 0

:=	: μ ref # μ → unit	R	(Assignment)

SPECIAL CONSTANTS (not identifiers)

† ()	: unit
† 0, 1, ~1, 2, ~2, ···	: int
† " --- "	: string

NOTES

All marked (†) are constants or constructors, so may appear in varstructs; "ref" is polymorphic, with type 'a → 'a ref, in varstructs only, "explode" yields a list of strings of length 1; "implode" is iterated concatenation (^) μ stands for any monotype, and γ is explained in Section 7.2.
All functions escape with their names on inappropriate arguments.