CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
LABORATOIRE d'ANALYSE et d'ARCHITECTURE des SYSTÈMES

7, Avenue du Colonel Roche, 31077 Toulouse Cedex, France
Tel: (+33) 61 33 62 00. Telex: LAASTSE.520930F. Telefax: (+33) 61 55 35 77

LCS USERS MANUAL

Version 5.0 of June1998

Release Notes
Language Reference
User Interfaces
Libraries

Bernard Berthomieu

Bernard.Berthomieu@laas.fr
tel: +33/0 5 61 33 63 63
fax: +33/0 5 61 33 64 11

LAAS/CNRS Report 98269, June 1998

Contents

1 Introduction

1.1 LCS

This document is a tentative reference manual for the language LCS. The language LCS is designed as an extension of Standard ML [Mil 90], by communicating agents based upon Robin Milner's Calculus of Communicating Systems, introduced in [Mil 80] and refined in [Mil89]. Besides agents, LCS differs from SML by some extensions and also some omissions.

Omissions include records of the core language and functors and signatures of the module system, but products and structures are supported. The reason for omission of functors and signatures is that features of the type system of LCS require extensions of these concepts that we are still investigating. On another hand, the omission of records is rather a matter of syntax; it was felt that the concrete syntax of LCS (which supports behavior expressions and behavior types) was already rich enough so that addition of records and record types might have confused the users. As SML records can be encoded into products, records were just omitted.

The main extension is undoubtedly behaviors, but there are a few other extensions, either resulting from introduction of behaviors and motivated by a better integration of ML and CCS, or independent from behaviors. Extensions dependent upon behaviors are essentially an enrichment of the type system of SML with so-called "behavior types", technically similar to extensible record types. Another dependent extension is the enrichment of expressions allowed to be recursively defined. Independent extensions include an available-to-users overloading mechanism, a more permissive than standard definition of equality types, and lazy values. The treatment of imperative types also differs from the standard one, it is based on so-called "weakness levels"; weakness annotations are exclusively handled by the compiler though (thanks to the omission of functors), and there are no imperative type variables. The several sorts of type variables found in the type system of LCS are all "first-class" (the usual ones, plus equality type variables and behavior type variables). The type system of LCS is order-sorted, with an inclusion relation linking the different sorts of types.

Agents capabilities extend those offered by CCS. Let us recall that CCS allows one to express behaviors, which may be seen as process scripts. Simple behaviors can be composed to build more complex behaviors. Interprocess communications are done through named ports, following the rendez-vous discipline. Recursion, combined with port relabelling features, allow to express cleanly and concisely complex and dynamic communication structures. LCS implements all the features of CCS, including value passing and recursion, plus some derived constructions and several strong extensions. The extensions are mainly a richer kind of communication ports, that permits users to compute ports in some sense, support of side-effects, higher order (behaviors are first-class values) and a process-level exception mechanism (events). The typing discipline for LCS agents extends that of ML; LCS is strongly typed language with polymorphic types, even for processes.

1.2 This manual

LCS is conveniently introduced as an extension of SML with CCS based behaviors, though everything was made to make behaviors not appear as an added feature. The LCS execution interfaces behave more like a "shell" program than an usual SML top-level, with processes as the highest level computation devices. Nevertheless, for ease of explanation, behaviors are presented in this document separately from the rest of the language.

This language reference manual contains four sections following this introduction. The features of the language are presented in the next two sections. Section 2, entitled "the functional subset", discusses actually all the features of the language, except those related with behaviors. At the present time, this section assumes knowledge of SML, as appears in [Mil 90], and concentrates on the differences between it and the functional subset of LCS (besides agents). Readers familiar with SML may safely skip this section on first reading, the others are prompted to read at least some introductory material about SML before going further (a list of possibly useful references is provided at the end of this document). Section 3 introduces agents and behaviors; some knowledge of CCS (as gained e.g. by reading [Mil 89]) might be helpful there for a deeper understanding of the features described. For reasons discussed above, sections 2 and 3 are slightly mutually recursive.

Sections4 and 5 overview typechecking and evaluation, respectively. A more detailed exposition of evaluation in LCS, as well as some good hints on its type system are found in reference [Nie 97]. Finally, an Appendix summarizes the syntax of the language.

In addition to the present *Language Reference* manual, reference informations include the *Release Notes*, the *User Interfaces* manual, and the *Libraries* manual.

Finally, it is out of the scope of this user manual to discuss the details of implementation of the language, or to constitute a tutorial on functional or concurrent programming.

Throughout the document, “standard” means “conforming to the definition of Standard ML”, as of 1986, that is as found in [Mil 90]. We have no plans yet to adopt the revision of SML known as SML’97 [Mil 97]; latter versions of LCS might provide both interfaces though.

2 The functional subset

2.1 Syntax

Microsyntax

Microsyntactic conventions are standard, except for the following:

- Identifiers in the type variable class may begin with "_"; no identifier, in any other class, may do so. Underbar prefixed identifiers are reserved for behavior type variables, a particular subset of types;

- A new syntactic class is introduced: port labels, concretely similar with variables;

- A new class of special constants is introduced: integers in multiple precision, which have type `mp`, distinct from the standard integer type `int`. `Mp` integers are represented by sequences of digits beginning with digit "`0`". Thus `00`, or `0123`, are parsed as multiple precision integer constants, while `0`, or `123`, are parsed as usual integer constants (in the range ~2^30 to 2^30 - 1 here);

- Characters in the range 161..255 are considered printable, they are allowed in identifiers and strings. In that range, those in subrange 161..191, as well as 215 and 247, are allowed in symbolic identifiers, the remaining are allowed in alphanumeric identifiers. Selection of the printable character set to use is controlled by primitive `System.extended_charset` (see the *Libraries* manual).

- Additional reserved words are:

```
agent, event, lazy, signal, catch, stop
/\, //\, /\\, //\\,
\/, \\/, \//, \\//,
->, #, ||, !, ?
overload, nonoverload
```

- The commands of the particular user interface in use are also considered reserved words; these are documented in the *User Interface* manual.

Expressions and declarations

All Standard ML features are supported except records, functors and signatures (signatures are accepted in structure declarations though, but considered comments). LCS expressions include thus all record-free, signature-free and functor-free SML expressions not using the new keywords above, and, in addition, the behavior expressions (discussed in Section 3).

Two new directives are introduced: **`overload`**, and **`nonoverload`**, which allow the user to bind several values to an identifier, as explained in the next section.

The LCS user interfaces also use a few commands which cannot be considered declarations, as their use is restricted to top-level. These are discussed in the *User Interface* manual.

The full set of syntax rules for LCS is given in the appendix.

2.2 Directives

Fixity Directives

(standard)

Overloading Directives

overloading rules

Standard ML uses overloaded identifiers to denote arithmetical functions operating on either integer or real numbers, but does not provide the user with the possibility of overloading arbitrary identifiers; LCS offers a flexible mechanism for doing so.

One may set or reset "overloading" status for an identifier, as one may set or reset "infix" status for it; two new directives are provided for this purpose, each one applying to a list of variable identifiers:

```
overload id1 ... idn
nonoverload id1 ... idn
```

Overloading directives may have top-level or local scope, as fixity directives do. Let us say that an identifier has "overloading" status when it occurs in the scope of an **`overload`** directive for that identifier. When binding an identifier that has overloading status, a new version of it is defined (type and value) and added to the current bindings of this identifier in the environment rather than hiding it (in addition, its type must meet some conditions soon to be described). An identifier with several values bound to it is said "multivalued". Note that an identifier may have overloading status without being multivalued, or even without being bound at all; conversely, an identifier may be multivalued without having overloading status. When overloading status for an identifier is cancelled (by the **`nonoverload`** directive), all values bound to this identifier remains in use, but any new binding of that identifier will hide its previous bindings, as usual. Beside this, scope rules for identifiers are standard.

Two simple rules summarize overloading constraints:

(i) As in SML, the type of every non binding occurrence of a multivalued identifier (whether is has overloading status or not) must be unambiguously determined as one, and only one, of those assigned to that identifier; there is no implicit type coercion, type constraints may be used to help resolution if needed. Note that this implies that the types of the different values bound to an identifier must all be distinct; they are further restricted to be monotypes[1].

(ii) The different values bound to a multivalued identifier must be unambiguously distinguishable from their types. This constraint is implemented as: the different types of the versions bound to an overloaded identifier must be monotypes and pairwise distinct.

These rules are enforced by the type checker. Enforcement of the rules is deferred to the end of the typechecking pass of the current top-level command (so usage of the identifiers may occasionally avoid the use of explicit type constraints in bindings). For constraint (ii), binding a version that shares its type with an existing version results in hiding of the former version (only the last version with that type is kept). An exception to (i) above is admitted when an identifier which is currently bound to a (single) polymorphic value is given overloading status; overloading status is not refused for that variable, but no new versions of it would be accepted, as violating a posteriori rule (i).

Finally, only identifiers in the variables class may be overloaded (constants, constructors or types cannot be overloaded).

structures and overloading

Structures do not trap overloading status of identifiers (they do not trap fixity properties either). A structure only captures the versions of an overloaded identifier that are defined within the structure. When opening the structure (by the **`open`** directive), the results on bindings depend on the current overloading status of the identifiers bound in the structure: Those having overloading status in the current environment are added the ver-

1. The treatment could accommodate as well polymorphic values with pairwise-disjoint polytypes, but we are not sure the benefits would justify the heavier implementation effort, considering the limited aims of that form of overloading.

sions captured in the structure; those not having overloading status have their binding(s) replaced by those found in the structure.

examples

The infix symbol “+” is initially bound to integer addition, real addition and mp addition, and its overloading status is off, as that of all identifiers in the initial environment. Assume we wish symbol + to denote termwise addition of lists of integers too, and also addition of an integer to such a list. This is done by the declarations:

```
- overload +
= exception Length
= fun i + [] = [] | i + (h::t) = ((i:int) + (h:int))::(i + t)
= fun l + i = (i:int) + (l:int list)
= fun [] + [] = []
=    | [] + _ = raise Length
=    | _ + [] = raise Length
=    | (h1::t1) + (h2::t2) = ((h1:int) + (h2 :int)):: (t1 + t2)
= nonoverload +;
+ overloaded
warning: redeclaring +
warning: redeclaring +
warning: redeclaring +
+ nonoverloaded
exception Length : exn
val + = - : int *  int list -> int list
val + = - : int list *  int -> int list
val + = - : int list *  int list -> int list

- 100 + [1,2,3] + [34,12,56];
val it = [135,114,159] : int list
```

A warning is issued for every overloading binding, but only the type and value of the last bound version are printed, for every version added. Identifier + has now six values bound to it, of six different monomorphic types. One may notice that among the two instances of the symbol + that appear in the body of the third declaration, only one is a recursive call (the second occurrence); with the help of a type constraint, the first occurrence is identified as integer addition.

One obtains the set of types assigned to some identifier by causing a type error, as in:

```
- op +;
unresolved type for overloaded +
type:
'a
should be matched by one exactly of:
int * int -> int
real * real -> real
mp * mp -> mp
int * int list -> int list
int list * int -> int list
int list * int list -> int list
typecheck failed
```

As another example, the lcs libraries provide for all special constant types a `makestring` function, for obtaining a string representation of a value of this type, and a `print` function, for printing that representation. In the initial environment, functions `print` and `makestring` are overloaded at all special constant types, one may easily extend these to operate on other monotypes, as done above for operator +.

2.3 Built-in types

Supported

All standard types are supported, except record types, but including vectors, arrays, and bytearrays. Additional types are provided, these are (in the initial environment) the multiple precision integers (type mp), lazy types (discussed in the next section), windows (briefly discussed thereafter), and objects (or machine representation of LCS values, discussed in the *Libraries* manual). There are also specific types added, pertaining to the process extension, which are described in Section 3 of this document (types behaviors and event).

Streams and Windows

File input and output are performed through values of type `instream` and `outstream`, respectively. Those streams are created by calls to opening functions. Besides many additional library functions (see *Libraries*, structures `Dev`, `Io` and `Stdio`), the treatment of file input/output is standard. Types `instream`, `outstream`, as well as type `control` (soon to be discussed) are defined from a more basic `dev` type, also used to encode the lcs abstraction of TCP sockets discussed later in the document.

In addition to streams, LCS provides terminal windows. Windows are either predefined (`std_win`, the console window), or created by an `open_set_win` call or one of its derived forms (see structure `Io`). Only windows of "terminal" kind are currently provided, graphics windows might be provided later. Windows created through `open_set_win` calls are associated with physical windows on the user's terminal.

The window abstract type is the association of an `instream`, an `outstream`, a `control`, a `dev`, and a `string`. These components may be accessed by the `win_in`, `win_out`, `win_ctrl`, `win_frame`, and `win_term` primitives. The last two primitives return an lcs device encoding the physical window frame, and the terminfo string associated to the window. Characters are read from a window by reading the instream associated with it, and are written in a window through its outstream. Controls may be seen as particular kinds of streams, used for transmitting exceptional conditions. Primitive `wait_ctrl_win`, called with some window w as argument, returns a control sequence (a string) when that control sequence has been typed in the window. Only the "interrupt" sequence (`^C`) and the "kill" sequence (`^\`) are currently supported. The control mechanism allows one to attach a controlling window to a system of processes (see *Libraries*, structure `Beh`).

Finally, LCS distinguishes interactive streams (streams associated with a terminal or window), from other streams (associated with files). Functions `is_term_in` and `is_term_out` permit the user to distinguish interactive streams from others. Interactive streams are necessarily predefined or created as components of windows; they are closed by typing in a `^D` character or by a call of a closing function.

2.4 Lazy types

The type lift of lazy values

On some occasions, one may need to manipulate infinite data structures, e.g. infinite lists or trees. The constructions allowed in SML do not allow one to build such data structures simply; except by using functions, but these represent a poor way of encoding some infinite structures. Possibilities like this are offered by some programming languages (e.g. Miranda, Lazy ML, Haskell, etc.). The idea here is to provide such possibilities in a way that would preserve all other constructs of SML, with their usual syntax and semantics. This precludes, for instance, particular (lazy) interpretations of existing lists, or products.

Also, providing directly a type of lazy lists, for instance, would lack genericity (trees too might need to be lazy). The LCS solution is to provide lazy capabilities by a single, non strict, pseudo-constructor: `up`, of type `'a -> 'a lift`. Evaluating expression `up e` returns a value (a lazy value), but does not evaluate the sub-expression `e`:

```
- val ex = up (12 + 54);
val ex = up - : int lift
```

Evaluation of the content of a lazy value is obtained either by applying function `down` to it or through extraction of its content by pattern matching. Function down is defined by:

```
- fun down (up x) = x;
val down = - : 'a lift -> 'a

- (down ex, ex);
val it = (66, up 66) : int * int lift
```

Lazy lists, for instance, are easily built from type `lift` and existing types (see *Libraries*, structure `Lizt`).

Lazy values may be recursively defined, they may be declared with **`val`** (if non recursive) or with either **`val rec`** or **`lazy`** (if recursive). The body of a recursively defined lazy value must necessarily start with constructor “up”. Lazy values do not admit equality.

Operationally, one may see lazy values as function values (closures) with an additional effect: the implementation guarantees that the body of a lazy value is evaluated at most once, when it is first extracted. Further accesses of the content of a lazy value simply result in consulting a previously computed value.

Matching lazy values

Pattern matching is restricted for the `up` constructor. In the initial environment, non exhaustive “up” patterns are rejected by the compiler; this restriction guarantees that pattern matching has no side-effects. It can be turned off through primitive `refutable_lazy_pattern` (found in structure `System`); the compiler still flags refutable lazy patterns then, but accepts them.

```
- refutable_lazy_patterns true;
val it = () : unit

- fun check (up 12) = true;
warning:  up applied to non exhaustive pattern
warning:  match not exhaustive
(up 12) => ...
val check = - : int lift -> bool
```

Abstraction

The `up` constructor and the `down` function implement the `delay` and `force` functions discussed in e.g. [Hen 80]; the essential difference is that type checking here statically prevents a non-delayed value to be forced. It is a folklore result that type lift and constructor up can be approximated with available SML types and functions, with lazy values encoded as functions, and using a reference to update its contents when first “applied”. There are three obvious disagreements with this abstraction though, notably the fact that the use of a reference for holding the content of a lazy value implies a polymorphism restriction on it. In our treatment, the fact that the encapsulated reference is unreachable (embodied in the implementation of up) and updated only once allows lazy values to be fully polymorphic.

2.5 Equality

Example

Equality is not available at all types, but only at those which “admit equality”; a subset of types soon to be defined. Types admitting equality constitute a proper subset of all types; type variables standing for equality types begin with two primes, as in `''a`.

Though on the same ground, the definition of equality types in LCS is slightly more liberal than that used in SML. Let us illustrate the difference with an example:

Let datatype New be defined by:

```
datatype ('a,'b) New = N of 'a ref * 'b;
```

In SML, types `(''a,int)New`, or `(bool,int)New` admit equality, but types `('a -> 'a, int)New`, or `('a, int)New`, do not, as they are built from type `New` (which admits equality) and type `'a ->'a` , or 'a, respectively, (which do not). As a consequence, one could not compare for equality the value `(N(ref f,3))` with `(N(ref g,3)),` where `f` and `g` are arbitrary functions with same types, though one could compare for equality the pair `(ref f,3)` with `(ref g`, 3).

LCS considers type `('a -> 'a, int)New` as admitting equality, on the ground that whatever the instantiation of type `'a` in the declaration of type `New`, the types of all parameters of constructors of type `New` admit equality. Less specifically, type `(ty1,ty2)New` admits equality whenever the `ty2` parameter do, whatever the `ty1` parameter is.

Types admitting equality

Let say that a type is a constructing type for a type operator `tycon` iff, in the binding of `tycon`, some constructor has an argument of this type. Then the types admitting equality are defined as follows in LCS:

(1) Neither type `exn, object, instream, outstream, control` or `window`, nor any behavior type, nor any type built with type constructors `function` or `lift` admits equality;

(2) Equality type variables (double quoted type variables), special constant types `int`, `mp`, `real`, `string`, any type built with the `ref` type constructor, and any product of types admitting equality, admit equality;

(3) A type built from a user defined type operator `tycon` and argument types `tyargs` admits equality iff after replacing the formal type parameters in the definition of `tycon` by `tyargs`, all constructing types of `(tyargs)tycon` admit equality (equality being assumed for the types built from `tycon`, if any).

For type declarations, and in addition to normal detectable binding information, the compiler prints the most general type of values that admit equality and whose type is built with this type operator. This is the information printed between square brackets in the following example:

```
- datatype ('a,'b) New = N of 'a ref * 'b;
datatype ('a, 'b) New   [('a, ''b) New]
con N : 'a ref * 'b -> ('a, 'b) New
```

2.6 Recursion

Types admitting recursion

Not only functions, but also lazy values and behavior values may be recursively defined. The values defined in a recursive declaration must be one of:

- A function expression, built by **`fn;`**
- A behavior expression, built by **`stop`**, **`return`**, action prefixing (μ=> or **`signal`**), relabelling (- {P/Q}), restriction (- {/R}), or composition (`/\`, `/\\`, `//\`, `//\\`, `\/`, `\//`, `\\/`, `\\//`, or **`catch`**).
- A lazy expression, built by `up`;

Non recursive values of any kind can be declared by **`val`**. Values of different kinds declared mutually recursive must be declared by **`val rec`**. For homogeneous declarations, LCS provides three specialized declarators:

- **`fun`** may be used to declare functions (as in SML);
- **`agent`** may be used instead of **`val`** or **`val rec`** to declare behaviors; the body of the declaration must have the shape of a behavior expression then.

- **lazy** may be used instead of **val** or **val rec** to declare lazy values; the body of the declaration must have the shape of a lazy expression then.

2.7 User defined types

Example

Type and datatype bindings in LCS are compatible with SML in the sense that all type and datatype declarations following SML rules are legal LCS declarations, but LCS incorporates an extension of these bindings, motivated by the introduction of behaviors, and by the richer equality types.

The LCS types constitute an order-sorted set with three different sorts of type variables (and types): plain variables (`'a`,`'b`, etc.), substitutable by any type, equality variables (`''a`, `''b`, etc.), substitutable only by equality types, and behavior variables (`_a`, `_b`, etc.), substitutable only by behavior types. Any kind of variable may appear in an LCS sentence everywhere a type variable may appear in an SML sentence, including in formal parameters of type bindings.

As an example, one cannot declare in SML a constructor that takes only equality types as arguments (though this may be enforced at each use of the constructor, by a type constraint). When defining a type `alist` of association list, for instance, it might be wise to restrict the type of first elements of pairs in the lists (the keys) to admit equality; this can be expressed in LCS by defining type `alist` by:

```
datatype (''a, 'b) alist = null | cons of (''a * 'b) * (''a, 'b) alist;
```

Similarly, one can define homogeneous pairs of equality types by:

```
- type ''a pair = ''a * ''a;
type ''a pair
```

And in all of its occurrences, type constructor `pair` would be handled as a pair of identical equality types:

```
- (1,2) : int pair;
val it = (1,2) : int pair

- (Match,Bind): exn pair;
equality type required
instance of type: exn
should match:     ''a
typecheck failed
```

Though convenient, this extension to the type bindings does not seem to be essential in SML (its effects can be simulated by type constraints). But, as shown in the sequel, it is necessary in LCS.

Justification

(This paragraph may be skipped on first reading. We assume here that Section 3 has been read.)

Consider the following declaration of function `f`. As easily checked, `f` cannot be given a type in the SML typing system, for the reason that its type should solve the equation (`'a = 'b -> 'a`) which admit no solutions in this type system (SML has no recursive types):

```
- fun f x = f;
(ill-typed)
```

`However, an approximation of` function f can be defined by encapsulating the recursive call of `f` in a user-defined type, whose constructor would operate as a type conversion function, as follows:

```
- datatype 'a Fun = F of 'a -> 'a Fun;
datatype 'a Fun
```

```
con F : ('a -> 'a Fun) -> 'a Fun

- fun f x = F f;
val f : 'a -> 'a Fun
```

A similar problem arises with agents sending of receiving their own behavior as messages. The following agent, for instance, would not typecheck in LCS (its type should solve equation `_a = {load: _a}_b`):

```
- agent a = load? x => x;
(ill-typed)
```

The intent is obviously to "download" a behavior on some communication line `load`; and then operate as the behavior loaded. A first possible solution to this problem would be to hide the port through which the behavior was received in the subsequent calls of this behavior, by defining `a` by:

```
- agent a = load? x => x {/load};
val a = - : {load: _a}_a
```

But that solution would forbid the downloaded behavior x to communicate through port `load` again for, eventually, loading another behavior. Thus downloading could be performed once only through label `load`. The satisfying solution is similar to the function encapsulation solution in the previous example. Let us define a type `Agt` of agents loading behaviors on port `load` as follows:

```
- datatype _a Agt = A of {load: _a Agt}_a;
datatype _a Agt
con A : {load: _a Agt}_a -> _a Agt
```

Then, assuming behaviors are encapsulated with constructor A when sent over port `load`, the previous agent `a` may be written:

```
- agent a = load? A x => x;
val a = - : {load: _a Agt}_a
```

Defining datatype `Agt` requires a behavior type to be the constructing type of constructor A. The author cannot think of any way of solving this problem using only "plain" type variables as parameters of type constructor Agt, considering that plain type variables cannot be written in place of behavior variables.

The LCS treatment allows thus behavior variables to stand as formal parameters of type constructors, and equality variables too since this does not bring any additional complication.

2.8 Exceptions

The treatment is standard, except that handling polymorphic exceptions relies on weakness annotations for type variables (see Section 2.10).

The effects of raising exceptions from within behaviors are explained in section 3.9.

Section 3.8 discusses events, a process-level exception mechanism.

2.9 References

The treatment is standard, except that polymorphic references rely on weakness annotations (see Section 2.10).

2.10 Weak polymorphism

References, exceptions and events suffer a restriction on polymorphism. The reason is illustrated for references by the following example (but one could find similar examples involving exceptions or events).

```
let val s = ref [] in s := [1]; "" :: !s end;
```

If the type of value s was generalized as for regular values, it would have in its scope the generic type `(∀'a.'a list) ref`; permitting the obviously wrong treatment of s in the program above. To prevent this, an additional constraint is established for reference, exception and event values, which is that, when created, these values should always have a monotype. This constraint would be satisfied by restricting these values to have monomorphic types at each of their occurrences, but this would be too strong a restriction; it is only when these values are created that they need to satisfy the constraint.

The treatment adopted in LCS is similar to that used in New-Jersey SML 0.93. Type variables in the types of polymorphic references, exceptions and events are decorated with an integer (its "weakness level", which is at most equal to the number of abstractions suspending the creation of the reference, exception or event value. A weakness level of 0 mean that the value is created (it must have a monotype then). Weak type variables (type variables having a bounded weakness level) must be instantiated by monotypes before their weakness level reaches 0.

For this reason, the reference constructor `ref` has type `'1a -> '1a ref`. Some examples are:

```
- fun f x = let fun g x = ref [] in (g x, g x) end;
val f = - : 'a -> '1b list ref * '1c list ref
```
(* one abstraction delays the creation of two distinct references *)

```
- fun f x = (x, ref);
val f = - : 'a -> 'a * (2b -> '2b ref)
```
(* two abstractions delays the creation of reference *)

There is a difference, however, between the treatment of weak variables in LCS and the treatment of njsml 0.93: weakness annotations in LCS are exclusively handled by the typechecker. No type written by the users, at any place, need mention any "weakness level" or "imperative" constraint[1] (the parser would just remove them, anyway).

1. Of course, this would cause trouble if LCS supported functors.

3 Behaviors and Processes

3.1 The syntax of behaviors

Behavior expressions

The syntax of SML expressions is enriched of behavior expressions (actions, relabelling, compositions, etc.), with their own concrete syntax. Here is an excerpt of the grammar of LCS expressions showing the behavior constructions (the full syntax is given in appendix):

exp ::=

smlexp	standard SML expressions
bexp	behavior expressions

bexp ::=

stop	termination
exp ren	relabellings
{**do** exp} => exp	commitment action
port! {exp} => exp	output action
port? {pat} => exp	input action
signal exp {**with** exp}	signal event action
exp C exp	compositions, where C ε {\/, \\/, \//, \\//, /\, //\, /\\, //\\}
exp **catch** smatch	event handling
exp -> exp	guarded conditional

port ::=

label{#exp}

Agent declarations

Behaviors belong to the same syntactic class as expressions. Though behaviors may be declared by regular **val** or **val rec** declarations, a new derived declaration is introduced: **agent**. Behaviors may be recursively defined. The **agent** declaration, as **fun** and **lazy** are, is a specialized form of **val rec**; one may use **agent** only in declarations whose body is a behavior expression (an expression of class bexp above).

Here are some typical behavior declarations:

```
- agent a = p? => q! => stop;
val a = - : {p, q}_a

- agent b = p? => q! => b;
val b = - : {p, q}_a

- fun c x = p! x => q? y => (c x /\ c y {/q});
val c = - : 'a -> {p: 'a, q: 'a}_b

- fun d x y = if y>0 then p#x! y => d x (y+1) else stop;
val d = - : ''a -> int -> {p#''a: int}_b
```

The first behavior could be declared by **val** as well since it is not recursive. The second could be declared with **val rec**. The third and fourth cannot be declared by **agent** since their bodies are functions.

The syntax of types is enriched of “behavior types”, an infinite family of types. A behavior type consists of a behavior-type variable (or behavior variable for short), necessarily beginning with an underbar, possibly prefixed by a series of type assignments to communication labels, between curly brackets and separated by commas (see the types printed for the above values):

ty ::=
    <standard>
    {"{"lab1{#t11}{:ty12}, ..., labn{#tyn1}{:tyn2}"}"} btyvar

Each of these type assignments assigns two types with a label, both defaulting to type `unit` if absent. In type "`label{#ty1}{:ty2}`", type `ty1`, if present, is the type of the extension of ports with label `label`, and type `ty2` is the type of the messages that may be transmitted through ports with label `label` (see the fourth example above for an example of behavior type with extension and message types made precise). Assignment of types to behaviors will be discussed in Section 4.

Event declarations

Finally, in addition to behaviors, LCS introduces events, which play a role resembling that of exceptions, but meaningful at process level only. An **`event`** declaration declares one or several event constructors, possibly admitting a parameter, and possibly carrying a message. Here is an example:

```
- event E1
= and   E2 of int
= and   E3 with bool
= and   E4 of string list with bool -> int;
event E1 : unit evt
event E2 : int -> unit evt
event E3 : bool evt
event E4 : string list -> (bool -> int) evt
```

`E1` is a constant event constructor that carries the default message `():unit`); event constructor `E4` requires a parameter of type `string list` and carries messages of type `bool->int`. Raising an event is a particular action, explained in 3.3, event handling is discussed in Section 3.8.

3.2 Behaviors and processes

It is time now to make the differences between behaviors and processes most precise. Behaviors must not be confused with processes, though the purpose of behaviors is precisely that of defining processes. Behaviors are particular values, just like integers or functions. Evaluation of a behavior expression does not produce a process, but a behavior value, resembling a closure, or function value.

Processes are created from behavior values through particular process creation commands, discussed in length in the user interfaces manual (e.g. **`start`**, **`call`**). Starting a behavior creates a process and causes its recursive evaluation. Turning a behavior into a process is the only way of evaluating the body of the behavior. Starting a process from the execution interfaces creates a processes that runs in parallel with all other started processes, including the lcs top-level, itself running as a system of LCS processes.

3.3 Actions

Behaviors as action capabilities

Actions play a particular role in the semantics of behaviors; behavior expressions may be seen as denoting action capabilities.

The behavior which performs no action at all is denoted **`stop`**. The actions a behavior may perform are described thereafter. Behavior combinators (primitive or user defined) combine the action capabilities of several behaviors. The CCS theory upon which LCS is based is found in [Mil 80, Mil89]; LCS adds several strong extensions to the basic CCS formalism though.

There are four kinds of actions in LCS: commitment actions, input actions, output actions, and signal actions.

Commitment actions

exp ::= => exp
| {**do** exp} => exp

Commitment actions express internal moves of a process, possibly with a side effect. The behavior `=> a` denotes a simple internal move (the CCS τ move), followed by the behavior represented by `a`. The behavior **do** `exp => a`; expresses a commitment action with side effect, followed by a; an internal move is performed, then expression `exp` is evaluated for its side-effects and behavior proceeds as `a`.

Input and output actions

exp ::= port ! {exp} => exp
| port ? pat => exp

Input actions and output actions are communication actions. Output actions specify a port and optionally a message (default "`()`"). Input actions specify a port and optionally a reception pattern (default "`()`"). The message received in a communication is matched to that pattern, this eventually binds zero or more variables, or raises exception `Match` if no match is possible.

Action capabilities must be distinguished from actual communications or commitments. Evaluation at top-level of behavior `p!23=>`**stop**, or **do**`(r:=678)=>`**stop**, for instance, would not result in any action being performed, evaluation is thoroughly discussed in Section 5.

Signal actions

exp ::= **signal** exp {**with** exp}

Signal actions are relevant to event handling. Conversely to other actions, signal actions do not prefix any behavior, they are the last action of a behavior. A signal action "raises" an event, expressed as an expression of type `ty evt`, for some type `ty`, and optionally associates a message with it (specified after keyword **with**). All events carry a message, the default message is value `(): unit`); the type of the message passed with an event must be that of the argument of the `evt` type in use (i.e. `ty` if the event signalled has type `ty evt`). Events may be caught, as explained in the relevant section, catching an event communicates the message it brings to the event handler.

3.4 Compositions

Compositions

There are two families of compositions for behaviors: parallel compositions (corresponding to CCS "|") and ambiguity, or exclusive, compositions (corresponding to the CCS "+"). There are variants of the composition operators which are relevant to store inheritance, these are discussed in section 3.11.

Parallel compositions

Parallel composition enables two processes to communicate values each other. In terms of communication capabilities, a parallel composition has the interleaving of the communication capabilities of its components with, in addition, the capabilities resulting from communications between the components: internal communications appear to the context as commitments. Parallel composition is denoted by `/\`. E.g. behavior a below composes in parallel behaviors `p! "hello world"=>` **stop** and `p? x=>` **do** `(print x)=>` **stop**.

```
- agent a = p! "hello world"=> stop /\ p? x=> do (print x)=> stop;
val a = - : {p: string}_a
```

Exclusive compositions

A behavior may have the communication capabilities of some behavior or of some other, exclusively, determined by the first action performed in either component. The combinator that expresses this composition of capabilities is the "ambiguity" (or "sum") combinator, denoted `\/`. E.g.

```
- agent merge = i1? x=> oo! x=> merge \/ i2? x=> oo! x=> merge;
val merge = - : {i1?: 'a, i2?: 'a, oo!: 'a}_b
```

In the above example, decision between a sub-behavior or another is unspecified; the decision is not made locally but depends upon the context (the communication offers provided by other processes running in parallel with that defined from merge). If one wants to introduce such local non-determinism, he/she may use the commitment construct, as follows:

```
- agent merge = => i1? x=> oo! x=> merge \/ => i2? x=> oo! x=> merge;
val merge = - : {i1?: 'a, i2?: 'a, oo!: 'a}_b
```

In this version of merge, the decision of which port should be read, among i1 or i2, is still unspecified, but it is local to agent merge (nondeterministic, semantically). Note that local commitment decisions may introduce deadlocks if not properly used (think for instance of a behavior with a similar structure to merge above, offering values on ports i1 and i2 and running in parallel with merge).

Boolean guarded behaviors

Guarded behavior expressions are a derived form. They consist of two expressions separated by an arrow:

exp ::=
    exp1 -> exp2

The guarded expression `exp1 -> exp2` is typed and evaluated exactly as:

```
if exp1 then exp2 else stop
```

Guarded expressions are convenient for expressing alternative behaviors depending upon boolean conditions, as in:

```
(e1 -> b1) \/ ... \/ (en -> bn)
```

3.5 Recursively defined behaviors

Behaviors may be recursively defined (see Section 2.6). An agent taking parameters is not distinct from a function.

Note that the "guardedly-well-definedness" property, as coined in [Mil80], is not required for recursive behaviors. Evaluation of an unguarded and infinite sum of behaviors cannot be guaranteed to run without error at run-time, however. Some of these expressions may well run correctly, due to the effects of scheduling in the LCS virtual machine, but most of these would exhaust some physical resource of the machine, at run-time. Usage of such constructions is thus discouraged.

3.6 Communication ports

Communication ports

Ports consist of a label (in the class of identifiers) and an optional extension, or tag (in the class of expressions, default "()"). Tags are computed when the action is evaluated. The association of a tag with a port label constitutes a communication port. Tags must admit equality, but are otherwise of arbitrary type.

port ::=
    label {#exp}

Whenever possible, ports transmitting messages of different meanings (to the application) should be labelled distinctly. However, one can find applications requiring arbitrarily many distinctly labelled ports. As only finitely many port labels may appear in any behavior expression, one ought to use port extensions for handling these cases.

Relabelling

While port tags are regular values, which thus may be computed or passed as argument, port labels are just identifiers, which do not denote any value (they indirectly refer to channels, though). Labels have global scopes, unless otherwise specified. The scope of ports are controlled by hiding and renaming of port labels.

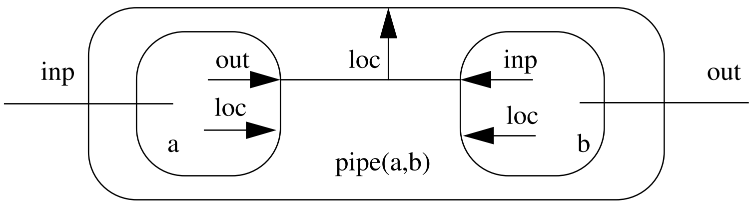
The renaming and hiding (or restriction) constructs are postfixed:

- A renaming has the form `b{new/old}`, where `new` and `old` are port labels. Its effect is to make in the surrounding context ports with label `old` renamed into ports with label `new`, and to hide the ports previously labelled `new`[1].
- A hiding has the shape of a renaming in which no "new" port is specified; hidings have the form `b{/res}`, in which `res` is the set of port labels to be hidden. Its effect is to make ports with labels in `res` unreachable from the surrounding context for communication purpose.

A hiding or renaming applies to all ports with the labels specified, whatever their tags are. Here is an example:

```
- fun pipe (a,b) = (a {loc/out} /\ b {loc/inp}) {/loc};
val pipe = - : {inp#''a: 'b, loc#''c: 'd, out#''e: 'f}_g *
               {inp#''e: 'f, loc#''h: 'i, out#''j: 'k}_g ->
               {inp#''a: 'b, loc#''l: 'm, out#''j: 'k}_g
```

Agent `pipe` (a behavior combinator) connects the `out` port of its first behavior parameter with the `inp` port of its second parameter, and makes this communication local. It is defined as the parallel composition of `a`, in which port `out` is renamed `loc`, and of b, in which port `inp` is renamed `loc`; port `loc` is then hidden, which makes it local to the composition.



The scopes of port labels in the pipe composition may be visualized as above. Circles in the picture delimit the scope of labels. Note that no hypotheses are made for the a and b behavior parameters about their actual communication ports; further, there is no way to enforce presence or absence of some ports in a behavior through a type constraint. In particular, a and b might have ports labelled `loc`, or might have no ports labelled `inp` or `out`. Renaming `out` as `loc` in behavior a and `inp` as `loc` in behavior b prevents however a and b from communicating directly through their previously existing `loc` ports if they had any; this explains the pattern above for label `loc`.

Port aliasing

Aliasing is a form of renaming that applies to ports, rather than labels. E.g. `b{p#34/q}`, `c{r/q#[3,4]}`, `d{r#5/r#12}` are aliasing[2], the first and second are equivalent to `b{p#34/q#()}` and `c{r#()/q#[3,4]}`, respectively.

1. This last hiding is what makes LCS renaming slightly different from CCS renaming.
2. Only the first form is implemented in LCS Version 5.0; other forms will be introduced in latter releases.

Port extensions need not be constants, they may be parameters as in the declaration:

```
- fun a b x y = b {p#x/q};
val a = - : {p#''a: 'b, q: 'c}_d -> ''e -> 'f -> {p#''e: 'c, q#''g: 'h}_d
```

Aliasing allows one to define behaviors using simple (i.e. non extended) local ports (which favors clarity efficiency of communications), and then to map them to arbitrary ports when the behavior is used in some context that requires that.

Equivalences

Hiding, Relabelling and Aliasing have form (exp ren) in which:

ren ::=
  {h1, ___ , hn}
  {r1, ___ , rn}
  {l1, ___ , ln}
h ::=
  / lab1 lab2 ___ labnn
r ::=
  lab11 lab12 ___ lab1n / lab21 lab22 ___ labnn
l ::=
  exp11 exp12 ___ exp1n / exp21 exp22 ___ expnn

Mixing hiding and/or renaming and/or aliasing in the same braces is not allowed. Hiding, renaming, and aliasing associate to the left; they may be multiple and/or simultaneous, with the following equivalences:

	a {/q1 q2 .. qn}	
eqv.	a {/q1, /q2, .. , /qn}	
eqv.	a {/q1} {/q2} .. {/qn}	
	a {p1 p2 .. pn/q1 q2 .. qn}	
eqv.	a {p1/q1, p2/q2, .., pn/qn}	
eqv.	a {p1/q1} {p2/q2} .. {pn/qn}	(if sets of labels (p)i and (q)i do not intersect)
	a {p1#e1 p2#e2 .. pn#en/q1#f1 q2#f2 .. qn#fn}	
eqv.	a {p1#e1/q1#f1, p2#e2/q2#f2, .., pn#en/qn#fn}	
eqv.	a {p1#e1/q1#f1} {p2#e2/q2#f2} .. {pn#en/qn#fn}	(if sets of labels (p)i and (q)i do not intersect)

3.7 Communication

Rendez-vous

Communications may occur between processes composed in parallel which offer matching actions. Two actions match if they have the same port specified, and if one is an input action and the other is an output action. Value (message) passing follows the rendez-vous discipline; input and output are performed synchronously.

If several, non independent, rendez-vous are simultaneously possible, then one is arbitrarily chosen and conflicting offers (if any) are retracted. Rendez-vous is the basic communication and synchronization mechanism in LCS.

Event passing (signal / catch)

Event passing, through the **signal/catch** mechanism discussed in the next section, implements another form of communication. Events may be seen as defining communication ports, a behavior raising an event passes the message associated with it to the handler specified for that event, if any (the message passed is that associated with the event by the **with** keyword). For ergonomic reasons, events do not appear as ports and have particular scope rules; further, event passing has the important additional side-effect of forcing termination of a subset of processes (those under the **catch** composition, as will be explained soon).

3.8 Events

What for

Exceptions raised within a process do not propagate outside that process. For this reason, we need a mechanism that allows some form of exception handling at process level, this is the main task of events and event handling. The kind of exceptions we would like to handle in communicating processes are asynchronous exceptional conditions related with communication, such as the failure of a process to successfully process a message. Exceptional conditions of this kind must usually be handled by restarting all or part of the system of communicating processes.

The desired semantics for such exceptions drove us to design an exception mechanism for handling such situations which is completely separated both from the normal communication mechanism and from the existing Standard ML exception mechanism. Events do not replace SML exceptions, the latter are still perfectly adequate for handling "functional" exceptions, but rather are complementary. Events have some analogies with Unix signals, for instance, but they may be dynamically created, they may vehicle messages, and they obey more flexible scope rules.

Events are built from event constructors, which are introduced by specific declarations (see Section 3.1). These constructors may be constant or parameterized. In addition, they may carry a message. A 1-ary event constructor with argument of type `ty1` and message of type `ty2` have type: `ty1 -> ty2 evt`; the type of messages appear as the argument of type `evt`.

Events need to be compared for equality (remember events identify particular communication ports), so the argument of a 1-ary event constructor must be of a type admitting equality; event messages do not suffer any type restrictions though. This property is enforced by the type checker when event constructors are declared. Finally, scope rules for events require events to be weakly polymorphic.

signal and catch

Events are raised by specific **signal** actions (see Section 3.3); handlers for events are specified by a **catch** construct, which is considered a behavior composition. An event handler is a sequence of catch-rules separated by "||". Every catch rule specifies an event (as an expression, not as a pattern), the keyword **with** follows the event, and is followed by a match selecting a behavior according to the message carried by the event. The simpler form `(event => behavior)` is admitted as a shorthand for (`event` **with** `_ => behavior`). Here is an example, from the events declared in 3.1:

```
fun A b x = b x catch
          E1 => b1
       || E3 with "" => b1 | "end" => b3
       || E4 (4*x) with m => b4 m;
```

A process defined by behavior `(A b x)` would run as the process with behavior `(b x)` until an event is signalled from within behavior (b x). When the event pops up, and if it is not caught within (b x), then that event is compared with those specified by the catch rules. If this event is `E1`, the behavior resumed is b1 (behavior b1 replaces b x). If the event is `E3`, with message `""`, or `"end"`, then the behavior resumed is `b2`, or `b3`, respectively; if the event is `E3` with another message, then event `E3` is caught anyway but the behavior resumed is (**signal** EXN **with** Match); in which EXN is a predefined event carrying exceptions as messages and used to propagate exceptions. Finally, if the event is equal to `E4 (4*x)`, then the message is bound to variable m and the behavior resumed is `(b4 m)`. Finally, if the event is neither `E1, E3` s, for any string s, nor `E4 (4*x)`, and if the behavior `(A b x) is` not under any catch construction that would provide a handler for that event, then the event offer becomes pending, making the process which raised it suspended forever (this is similar to a communication offer that cannot be matched).

If several event catchers are embedded, then all catchers, innermost to outermost, try to catch that event, until one eventually succeeds. Thus, events propagate "inside-out" only and catching an event affect only the processes under the particular **`catch`** that provided the handler.

Events are useful everywhere a group of processes may need be replaced by another (or terminated) on occurrence of some normal or exceptional condition. Contrarily to Unix signals, events cannot be used by a process to influence the behavior of another process arbitrarily related with the former, though such capabilities can be simulated with events and existing constructions.

Events vs. exceptions

An essential difference between an event and an exception is the behavior on absence of handler. While exceptions propagate on absence of handler, events do not; in absence of handler, raising an event only affects the behavior of the process that signalled the event. The reason why events in "catchers" appear as expressions rather than patterns (as exceptions do in exception handlers) is explainable by this difference. It should be noted too that catching an event may have irreversible consequences on the current system of processes, so, at process level, we just cannot catch an event then re-signal if it is not the intended event, as one could do with exceptions.

A predefined `EXN` event, carrying messages of type `exn` is provided. This event is automatically signalled when an exception is raised (and not caught) in the course of evaluation of a process; that process signals then event (`EXN` **`with`** `x`), where `x` is the uncaught exception. Installing a catcher for event `EXN` around a system of processes is particularly useful in its design phase, to analyze its failures.

3.9 Higher order

There are no restrictions on the type of messages a process may send or receive; these may be of any type, including behaviors, which make higher order expressions such as the following legal:

```
- agent b = load! A b => load? A c => c;
val b = - : {load: _a Agt}_a
```

Agent `b` above sends its own behavior on a port labelled `load`; then receives a behavior `c` on a port also labelled `load`, and behaves as `c`. Constructor `A` and type `Agt` are those discussed in Section 2.7.

Beside this, functions may take behaviors as parameters and produce behaviors as result. The LCS libraries provides a number of such derived behaviors combinators.

3.10 Light and Strong processes

Process stores

Any evaluation, at any place, may have a side-effect. The chapter on evaluation makes the evaluation rules precise. We concentrate here on scope rules for references. References values are defined as in SML, except that they have (dynamic) scopes.

Every process is attached with a store. A store maps reference values to their contents. The basic store assignment mechanism is inheritance; stores are inherited by processes when created at toplevel or through the composition constructions (`/\`, `\/` and **`catch`**), but one also has the possibility of assigning a private store to a process, obtained by a copy of its inherited store.

If the "light" process option is selected[1], then all processes share the same store, whatever the process creation command used and the process composition evaluated. If the "strong" process option is selected, then store assignment obey the following rules:

1. See the *Release Notes* or the lcs man pages for the available options of command `lcs`.

- When started, a process inherits the store of the top-level process, or of a copy of it, depending on the start command used (see *User Interfaces* manual for details);
- For parallel and ambiguity compositions, four pairs of predefined composition operators are provided. Stores are assigned as follows, depending on the combinators used:

light compositions	`/\`	(resp. `\/`)	both lhs and rhs inherit the current store
right-strong compositions	`/\\`	(resp. `\//`)	lhs inherits current store, rhs inherits a copy of it
left-strong compositions	`//\`	(resp. `\\/`)	rhs inherits current store, lhs inherits a copy of it
strong compositions	`//\\`	(resp. `\\//`)	both lhs and rhs inherit copies of the current store

- The **`catch`** construct acts as a light composition; the store associated with the process under **`catch`** and with the handlers is the store which was current when the catch construction was executed.

For example, the current store being `S0`, and after evaluation of all compositions in the following process (and assuming store `S0` is not updated by any concurrent process), processes `b` and `c` inherit store `S0`; process `a` gets a private copy of `S0`, and processes `d` and `e` share another copy of `S0`.

```
(a //\ b) /\ (c \// (d /\ e))
```

Message cloning

A process may freely pass to another messages having captured references. When the sending and receiving processes do not share the same store, those captured references are replaced at reception of the message by new references created in store of the receiver; in that case, the message is said cloned. Passing several times the same message to a process not sharing the store of the sender produces several distinct clones of the message in the receiver's context.

Concurrent assignments

The problem addressed here is that of protecting, or not, references from the well known concurrent assignment problem in systems of concurrent processes sharing their store.

LCS does not enforce any rule to prevent the well known concurrent assignment problem between processes sharing the same store. E.g. assuming reference `r` holds 1, then after execution of (**`do`** `(r := !r + 1) =>` **`stop`** `/\` **`do`** `(r := 4) =>` **`stop`**), the content of `r` may be one of 4, 5, or 2 (rather than the expected 4 or 5).

3.11 Processes vs Stream Input / Output

Stream based input/output coexist in LCS with CCS' style "behavioral" input/output, essentially for compatibility with SML. The coexistence of these two different input/output mechanisms may occasionally disturb the user by making organization of some programs a subtle task. Structure `Pio`, found in libraries, suggests "behavioral" interfaces for frequent file and window based input/output operations.

3.12 Streams and Distributed Programming

Distributed programming capabilities are provided through "network abstractions" (see the *Libraries* manual, structure `Network`, for details). Those network abstractions are implemented from Unix TCP sockets (structure `Network.Socket`) and a package implementing on top of them secured communication of structured data between lcs machines (structure `Network.Remote`). They provide "behavioral" interfaces for type-safe communications between lcs applications.

These distributed programming facilities must be considered experimental. Though their interfaces should be preserved, their implementation may significantly change in future releases.

4 Typing

The type system of LCS is explained in more details in reference [Nie 97]. The following informal explanations should suffice for most practical purposes.

4.1 Typing of functional values

Beside what follows, typing of functional values is standard.

Imperative type variables

These are removed, as mentioned in Section 2. Weakness annotations are introduced and handled by the compiler, they are shown in the types inferred for the values declared at toplevel, but the user may not ascribe any.

Equality type variables and equality types

Equality type variables are "first-class" type variables, as mentioned in Section 2. Type parameters of type operators may be equality variables (or behavior type variables, which also are first-class). Inference of equality types is slightly more liberal than in SML, as explained in Section 2 too.

Type constraints

Type constraints are handled more liberally than in SML. It is not required that type constraints exactly matches the type of the value it constrains, but simply that the constraint and the type of the value are unifiable (e.g. both expressions `2:'a` and `[]:int list` are accepted in LCS, while SML only accepts the latter); strengthening of constraints are flagged by the compiler, however.

4.2 Typing of behaviors

Behavior types

Behaviors have specific types in an infinite family of "behavior types". They have shape:

```
{p1#t11 : t12, ... ; pn#tn1 : tn2}bv
```

Also written:

```
{p1#t11 : t12} ... {pn#tn1 : tn2}bv
```

In which `p1...pn` are labels, `ti2` is the type of extension of label `pi, ti1` is the type of messages possibly passed through label `pi`. and `bv` is a behavior type variable (_a, _b, etc). Type assignments to labels (between curly braces) may be written in any order.

As typing devices, behavior types are similar (but not exactly equivalent) to extensible record types, or the so called row types. They can be interpreted as follows:

- Behavior type variables, or behavior variables for short, stand for unconstrained behavior types, that is they (implicitly) assign a pair of types variables with every label (one for the extensions, and another for the messages). Different occurrences of the same behavior variable in a type or in the type context denote the same type assignment to labels though, as in the types of compositions: `_a * _a -> _a`. Different type variables denote unrelated unconstrained behavior types (not sharing any variable).
- The behavior type in which label `p` has extension of type `t1` and passes messages of type `t2`, and in which other labels have the same type as in the behavior type ty is written `{p#t1:t2}ty`.
- In types `b1={p#t11:t12}_b` and `b2={p#t21:t22}_b` (assumed unquantified and found in the same context), every label has the same type in both `b1` and `b2`, except label `p` which has extension type `t11` and message type `t12` in `b1,` and extension type `t21` and message type `t22` in `b2`.

- In expressions where labels are assigned several types (e.g. `{p#t:t'}_a`, following from the interpretation of behavior variables), the leftmost type assignment for some label supersedes all others for that label (i.e. the types assigned to `p` by `_a` in this example is hidden by that assigned to `p` by the prefix `{p#t:t'}`).

The concepts of substitution (of a variable by a term), instantiation (of free variables), of generic instance (instantiation of bound variables), and of unification, can be extended to types containing behavior variables; provided behavior variables are substituted by behavior types only.

Typechecking

Types are assigned to all sub-expressions in a context that assigns a type to each identifier. Type assignment for functional expressions obey the standard rules; the typing rules for behaviors are summarized below:

- All occurrences of an identifier bound by abstraction or by input must receive, in their scope, the same type. Occurrences of locally or top-level declared identifiers receive as types generic instances of the types assigned in their declaration;
- **`stop`**, compositions (`/\, \/, etc.`), `signal` behaviors are typed as occurrences of top-level bound identifiers with types `_a`, `_a * _a -> _a`, and `_a`, respectively;
- In actions (except **`signal`** actions), the suffix behavior must have a behavior type. For input and output actions, the types assigned to the label occurring in the action, if any, must be the same that the types assigned to that label in the type of the suffix behavior;
- In a hiding, the type constraints assigned to the labels being hidden, if any, are relaxed in the result type. e.g. in "**`fun`** `A a = a {/p}`", `A` has type `{p#''a:'a}_a -> {p#''b:'b}_a`.
- In a renaming, the destination label gets the type of the source label. e.g. in "**`fun`** `A a = a {q/p}`", `A` has type `{p#''a: 'b}_c -> {q#''a: 'b}_c`.
- In a catch construct, all behaviors specified in handlers, and the behavior under the **`catch`**, must have same (behavior) type. Further the event expressions must have type `ty evt`, for some type `ty` (but events in different catch rules may have different message types).

polymorphism

A consequence of the definition of behavior types as infinite type assignments is that all behaviors always have polymorphic types since no program may require type constraints on an infinity of labels. If no means of defining "monomorphic" behavior types was provided, this would preclude in most cases building references to behaviors, or defining exceptions taking behaviors as parameters, or defining events passing behaviors as messages.

For this reason, two special type constants are introduced, considered monotypes:

- A constant type operator written `void`, which can only be assigned to a value by a type constraint.
- The type "_", which is the behavior type assigning type `void` with every port label.

Using type "_", we can now express monomorphic behavior types like `{q#bool:int list,r:real}_` or `{p:int}_`.

normal and compact forms for behavior types

Several forms may be used for representing behavior types. From their definition, the following four types, for instance (in which all variables are assumed universally quantified) are equal (remember the "leftmost assignment" rule above):

(1) `{p#''a:'a, q#t1:t2, s#''d:'d, r#''c:'c}_a -> {p#t3:t4, r#''b:'b,s#''d:'d}_a`

(2) `{q#t1:t2}_a -> {p#t3:t4, r:''b:'b}_a`

(3) `{q#t1:t2,r#''c:'c}_a -> {p#t3:t4}_a`

(4) `{p#''a:'a, q#t1:t2, r#''d:'d}_a -> {p#t3:t4, q#''e:'e, r:''b:'b}_a`

Type (1) is neither factorized, nor balanced. Type (2) is factorized, in the sense that the number of type constraints made explicit for labels is minimal. Type (3) is another possible factorization of type (1); it shows that the factorized form is not unique for a given type expression. There is a natural ordering among these, tough, the ordering chosen in lcs identifies (2) as the *canonical* form for types (1) to (4). Type (4) has been first factorized, and then balanced; in (4), all occurrences of a behavior type variable appear after a prefix which holds constraints for the same set of labels; type (4) is called the *normal* form.

Two options are provided by the LCS typechecker for controlling the printing of behavior types. The first option, provided as function `factorize_types`, enables or disables the factorization of multiple type assignments to labels. This option is initially enabled and should normally remain so; it is essentially provided as a debugging aid for the LCS developers. The second option might be of interest to users; it operates only if the first option is enabled and controls balancing of behavior types. “`balance_types true;`” makes behavior types in type expressions printed in the *normal* form (as (4) above), rather than in the shorter *canonical* form (as (2) above). This representation might help capturing the consequences of a type, in some occasions, though it generally makes type representations much larger.

types and sorts

Behavior types must not be confused with the sorts discussed in the CCS literature (see e.g. [Mil 80]). Sorts, there, are the set of port labels through which some behavior may communicate, associated with their polarity (input or output). The behavior types inferred by LCS do not vehicle any sort information; the case is typically illustrated by the type of the `pipe` combinator defined in Section 3.6: this type does not require that the composed behaviors have indeed ports labelled `inp` or `out` in their sorts. Behavior types must be understood as constraints that guarantee type-safe communications, in the case some are possible.

5 Evaluation

Evaluation of LCS V5.0 programs is explained in details in reference [Nie 97]. The following informal explanations should suffice for most practical purposes.

5.1 Reduction and expansion

Evaluation in LCS is applicative. Evaluation of functional expressions is standard; behavior constructs evaluate as closures, similar to function values. The process of evaluating an LCS expression (of any type) to produce a value is called *reduction*. On another hand, the body of a behavior value (process constructions) is evaluated upon a process creation request at top-level (through one of the commands explained in the *User Interfaces* manual), or, recursively, when the behavior closure is invoked from a running process. The process of evaluating behavior constructions (starting from a behavior value) is called *expansion*. Expansions produce expressions (one or several) which are in turn evaluated, and so on. Evaluation of an LCS process is typically an alternation of reductions and expansions.

Behavior values capture the current environment, together with the (code generated for the) body of the behavior. Creating a process from a behavior value first consists of attaching some information with the closure; this information is made of a “store”, a “position information”, which encode the concurrency and preemption status of the new process among the system of running processes, and a “communication environment”, which associates with every port reachable from that process the queues of processes waiting for input and of those waiting for output through this port.

5.2 Reduction

Context

Reduction affects the environment and store of a process. The environment maps identifiers to their values, it is extended by value bindings. The store maps references to their contents. Remember that stores here may be shared by several processes. Stores are extended by `ref` constructions and updated by assignment.

Reduction

Reduction of functional expressions is standard.

Lazy values encapsulate a boolean (the status) and a value (the contents). Up constructions reduce as pairs (status=`unevaluated`,`closure`), and are updated into (status=`evaluated`,`value`) on the first access to the contents, where `value` is the result of resuming `closure`.

All behavior constructions reduce as closures, capturing the current environment and the body of the construction. These constructions are:

- Actions, of all kinds;
- Compositions, of all kinds;
- The **`catch`** construction;
- Hiding and renaming;
- **`stop`**.

Reducing a renaming construction or an action construction that make use of extended communication ports evaluates the port extensions before the behavior value is produces. Similarly, reducing a **`signal`** construction reduces the events involved.

5.3 Expansion of behaviors

Context

Expansion, or evaluation of behavior constructions, takes place in the context of an environment E, a store S, a position information P, and a communication environment L. The position information of two processes, not precisely described here, allows to decide whether these processes are concurrent, or whether one is preempted or not on an action by the other. The communication environment L maps every possible port into two queues of processes, the queue of processes waiting for input on this port, and the queue of processes waiting for output on this port.

The communication environment L is private to processes, as is the value environment E, but the queues mapped to ports by L are shared by all processes which may reach them. Further, as already mentioned, the store S may be shared by several processes, and is thus subject to change at any time for reasons not related to reduction or expansion of the current process.

Expansion of actions

Expansion of a **`commitment`** action (a **`do`** action) consists first of terminating the processes conflicting with the current process. In effects, this updates the position information of all processes. Then, the expression associated with the commitment is evaluated for its side-effects, and the behavior following the action is evaluated.

For expansion of an **`input`** action, a process offering a matching communication on an equal port is first searched in the communication environment. If none is found, then the current process is suspended in the communication environment, in the input queue of the entry associated with the port involved; this updates L for other processes. If some process matching the communication offer is found (it must be a process which is concurrent with the current and found in the output queue of the entry associated with the port in L), then the matching process is removed from the queue, both processes perform a move (forcing termination of conflicting processes and update of positions) and the message encapsulated in the output suspension. Both sender and receiver are then resumed. If sender and receive do not share the same store, the message is cloned at reception, in the store of the receiver.

Expansion of an **`output`** consists first of reducing the message to be sent, expansion then proceeds symmetrically to expansion of input actions.

Expansion of a **`signal`** action first consists of reducing the message associated with the event. A matching handler is then looked for. If a handler for that event is found, then all processes protected by the handler are terminated, the message carried by the event is passed to the handler, and the handler is resumed. If none is found, then the signalling process is suspended (which is equivalent here to termination since event handlers cannot be installed for (running) processes).

Warning: Expansion of actions of several processes may overlap. Since, in addition, several processes may share a store, and expansion of actions may have side-effects, the result of expanding a set of actions concurrently may not coincide with that of expanding them sequentially (in other words, side-effects and store sharing break the CCS “interleaving” semantics).

Expansion of relabelling

Hiding and renaming affect the communication environment L.

`Hiding` of a label p has the effect of extending L with entries so that all ports that have p as label are now associated with pairs of new, empty, queues.

`Renaming` of a label p by a label q has the simultaneous effect of hiding p, and of associating with label q the queues formerly attached with label p.

`Aliasing` has an effect similar to a renaming, except it affects a single port rather than a set of them.

Expansion of compositions

Parallel compositions create two parallel processes from the current one. These processes inherit the current position, communication environment, and, depending on the composition operator used, either the current store, or a copy of it. Once these processes are created, their bodies are evaluated.

Ambiguity compositions are evaluated as parallel compositions except that they create conflicting (non concurrent) processes, this updates position information P.

Process creation commands (see the *User Interfaces* manual) are evaluated as parallel compositions, one of the composed processes being the top-level itself. Depending on the process creation command used, the process created inherits either the store of the top-level process or a copy of it.

Finally, the **`catch`** construction is expanded by first reducing all events in the catch rules, from first to last. Then the handlers are reduced, producing as many (suspended) processes as there are handlers, and the behavior protected by the **`catch`** is evaluated.

Process pruning

It was already mentioned, when introducing the “position” information, that an action performed by some process may force termination of an arbitrary number of other processes. LCS has a built-in mechanism, called “process-pruning”, that transparently removes such preempted processes from all process queues they may reside into.

6 Design notes

Design decisions

Several aspects of the LCS language must still be considered in experimental shape and subject to improvement in later releases (feedback from the users is expected here); these are mainly the following:

- Treatment of references: The scope of references is currently determined by behavior composition operators. There might be more flexible manners of controlling the scope of references, e.g. by using several kinds of reference constructors, determining the scope of the particular locations created.
- Atomicity of actions: Evaluation of several actions may overlap. Further, some actions may be partially evaluated, but never completed (because the process they belong to has been terminated as the result of expansion of a sum or catch composition). The necessity, or not, of enforcing some kind of atomicity of actions has to be investigated.
- The coexistence of two different input/output mechanisms: The availability of both stream-based and behavioral input/output mechanisms, may occasionally cause some apparently strange (but always explainable) scheduling effects. For homogeneity, the libraries provide behavioral interfaces for the usual stream input/output actions, allowing to handle a file or an interacting window like a process (see the *Libraries* manual, structure `Pio`).
- Module system. LCS Version 5.0 supports structures, but neither signatures not functors. Integration of behavior types with the signature matching mechanism of SML is being investigated.
- The current implementations of LCS V5.0 does not support port extensions that capture references, using such extensions can make an application crash. This is an implementation problem rather than a design problem though, having to do with the algorithm retained for comparing port extensions for equality. Several fixes are being investigated.

References

ML and Standard ML

[Gor 79] M.J. Gordon, A.J. Milner, C.P. Wadsworth, "Edinburgh LCF", LNCS 78, Springer Verlag, 1979, ISBN 3-540-09724-4 or 0-387-09724-4.

[Mil 90] R. Milner, Mads Tofte and Robert Harper, "The Definition of Standard ML", MIT Press, 1990. ISBN 0-262-63132-6.

[Mil 91] R. Milner, Mads Tofte, "Commentary on Standard ML", MIT Press, 1991. ISBN 0-262-63137-7.

[Mil 97] R. Milner, Mads Tofte, Robert Harper and D. McQueen "The Definition of Standard ML (Revised)", MIT Press, 1997. ISBN 0-262-63181-4.

Calculus of Communicating Systems

[Mil 80] R. Milner, "A Calculus of Communicating Systems", LNCS 92, Springer Verlag, 1980, ISBN 3-540-10235-3 or 0-387-10235-3.

[Mil 89] R. Milner, "Communication and Concurrency", Prentice-Hall International Series in Computer Science, 1989, ISBN 0-13-114984-9.

ML with Concurrency

[Nie 97] F. Nielson (Editor). "ML with Concurrency, Design, Implementation and Application", Springer Verlag Monographs in Computer Science, 1997, ISBN 0-387-94875-9.

(Includes sections on Concurrent ML, Poly/ML, Facile, LCS, and theory of concurrency).

Functional programming

[Bur 76] W.H. Burge, "Recursive Programming Techniques", The System Programming Series, Addison Wesley, 1975, ISBN 0-201-14450-6.

[Hen 80] P. Henderson, "Functional Programming: Application and Implementation", Prentice-Hall International Series in Computer Science, 1980, ISBN 0-13-331579-7.

[Wik 87] A. Wikstrom, "Functional Programming Using Standard ML". Prentice-Hall International Series in Computer Science, 1987, ISBN 0-13-331661-0.

[Bir 88] R. Bird, P. Wadler, "Introduction to Functional Programming". Prentice-Hall International Series in Computer Science, 1988, ISBN 0-471-01650-0.

[Rea 89] C. Reade, "Elements of Functional Programmimg", Addison Wesley, International COmputer Science Series, 1989, ISBN 0-201-12915-9.

[Pau 91] L. C. Paulson, "ML for the Working Programmer", Cambridge University Press, 1991, ISBN 0-521-39022-2.

Related notations for high level concurrent programming

[Hoa 85] C.A.R. Hoare, "Communicating Sequential Processes". Prentice-Hall International Series in Computer Science, 1985, ISBN 0-13-153271-5.

[Agh 86] G. Agha, "Actors, A Model of Concurrent Computation in Distributed Systems", MIT Press Series in Artificial Intelligence, 1986, ISBN 0-262-01092-5.

[Hen 88] M. Hennessy, "Algebraic Theory of Processes", MIT Press Series in the Foundations of Computing, 1988, ISBN 0-262-08171-7.

A1 Syntax summary

A1.1 Microsyntax

reserved words

The following are the reserved words. They may not be used as identifier.

abstype agent and andalso as case catch datatype do else end event exception fn fun handle if in infix infixr lazy let local nonfix nonoverload of op open orelse overload raise rec signal stop then type val with withtype while

( ) [ ] { } , : ; = => _ | || # ! ? -> /\ //\ /\\ //\\ \/ \\/ \// \\//

Independently from the reserved words of the language, there are additional reserved words, specific to the user interface in use. These are the following, for the executive user interface, and for the simulator user interface, respectively:

start, lstart, call, lcall, return, kill, reboot

try xpand run scan more steps commits right left rmost lmost tag jump search dead

special constants

There are four classes of special constants:

INT	integer constants
MP	multiple precision integer constants
REAL	real constants
STRING	string constants

An integer constant is either “0”, or “~0”, or else any non-empty sequence of digits not beginning with 0, possibly preceded by a negation symbol.

A multiple precision integer constant is either “00”, or “~00”, or else any non-empty sequence of digits beginning with 0, possibly preceded by a negation symbol. The leading “0” distinguishes mp integers from int integers.

A real constant is an integer constant, possibly followed by a point (.) and one or more digits, possibly followed by an exponent symbol (E) and an integer constant; at least one of the optional parts must occur.

A string constant is a sequence, between quotes ("), of zero or more printable characters, spaces or escape sequences. Each escape sequence is introduced by the escape character \, and stands for a character sequence. The allowed escape sequences are as follows (all other uses of \ being incorrect):

\n	A single character interpreted by the system as end-of-line.
\t	Tab.
\^c	The control character c, for any appropriate c.
\ddd	The single character with ASCII code ddd (3 decimal digits).
\""	"
\\	\

\f ___f\ This sequence is ignored, where \f __ f\ stands for a sequence of one or more formatting characters (a subset of non-printable character constituted of space, tab, newline). This allows one to write long strings on more than one line, by writing \ at the end of one line and at the start of the next.

identifiers

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

! % & $ + - / : < = > ? @ \ ~ ‘ ^ | *

If the `extended_charset` option is enabled (see library System), then characters with ascii codes 161 to 191, plus 215 and 247, are also allowed in symbolic identifiers, and those range 192 to 255, except 215 and 247, are allowed in alphanumeric identifiers.

In either case, reserved words are excluded. Exceptions to this rule are the symbol =, which stands for the equality predicate =, the boolean connectives (`orelse, andalso`) and the family of behavior composition symbols (`/\, /\\, ..., //\\`); these are reserved word and are also allowed as identifiers. These identifiers, however, may not be rebound; this precludes any syntactic ambiguity.

Identifiers are used to stand for seven different syntax classes, handled as six different name spaces:

STR	structure variables.
VAR, CON	value variables and constructors, including value, exception and event constructors.
TYVAR	type variables.
TYCON	type constructors.
BTYVAR	behavior type variables.
LAB	communication labels.

A type variable (TYVAR) is any alphanumeric identifier starting with a prime. A behavior type variable (BTYVAR) is any alphanumeric identifier starting with an underbar. The other five classes of identifiers are represented by identifiers neither starting with a prime, nor with an underbar.

The type variable class is further partitioned into “equality type variables”, starting with two primes, and “plain type variables”, with only one leading prime.

The seven syntax classes occupy six different name spaces, which means that a same identifier cannot denote both a constructor and a value if their scope intersect; but may denote, e.g. a structure, a value, a type constructor and a communication label. The syntactic class of an identifier is determined from its context.

comments

A comment (outside a string constant) is any character sequence within comment brackets (* *) in which comment brackets are properly nested.

lexical analysis

Each item of lexical analysis is either a reserved word, a special constant or an identifier. Comments and formatting characters separate items. At each stage the longest item is taken. As a consequence, spaces or parentheses are needed sometimes to separate identifiers and reserved words (as in, e.g. a := !b, rather than a :=!b).

A1.2 Syntax equations

bnf rules

Curly brackets { } enclose optional phrases, except when surrounded with double quotes, in which case they denote the actual concrete symbol.

Repetition of at least k iterated phrases with separator x is represented by "ph1 x __k__ x phn".

Alternatives are listed in order of decreasing precedence.

L (resp. R) means left (resp. right) association, where meaningful.

programs

lcs ::=	lcs program
dec ; lcs	declaration, (1)
cmd ; lcs	command, (2)
cmd ::=	user interface command (3)
...	

declarations

dec ::=	declaration
val vb	value declaration
fun fb	function declaration
lazy lb	lazy value declaration (7)
agent ab	agent declaration (8)
type tb	type declaration
datatype db	datatype declaration
abstype db **with** dec **end**	abstract type declaration
exception exb	exception declaration
event evb	event declaration
local dec **in** dec' **end**	local declaration
structure sb	structure declaration (12)
exp	expression, (9)
dir	directive
dec1 {;} __0__ {;} decn	declaration sequence
dir ::=	directive
infix {d} id1 __1__ idn	declare left associative infix, 0≤d≤9
infixr {d} id1 __1__ idn	declare right associative infix, 0≤d≤9
nonfix id1 __1__ idn	cancel infix
overload id1 __1__ idn	declare overloading
nonoverload id1 __1__ idn	cancel overloading
sb ::=	structure binding
var = **struct** dec **end**	simple
sb1 **and** __2__ **and** sbn	multiple
vb ::=	value binding
pat = exp	simple
vb1 **and** __2__ **and** vbn	multiple
rec vb	recursive, (4)
fb ::=	function binding
cl1	__1__	clm	clausal form
fb1 **and** __2__ **and** fbn	multiple
cl ::=	function clause
{**op**}var apat1 __1__ apatn{:ty} = exp	clause (5)

lb ::=	lazy value binding
var {:ty} = exp	simple (4)
lb1 **and** 2 **and** lbn	multiple
ab ::=	agent binding
var {:ty} = exp	simple (4)
ab1 **and** 2 **and** abn	multiple
tb ::=	type binding
tycon = ty	0-ary
tyvar tycon = ty	1-ary
(tyvar1; 1 ; tyvarn) tycon = ty	n-ary
tb1 **and** 2 **and** tbn	multiple
db ::=	datatype binding
tycon = constrs	0-ary
tyvar tycon = constrs	1-ary
(tyvar1; 1 ; tyvarn) tycon = constrs	n-ary
tb1 **and** 2 **and** tbn	multiple
db **withtype** tb	with abbreviations
constrs ::=	constructors
{**op**}con1{**of** ty1} | 1 | {**op**}conn{**of** tyn}	
exb ::=	exception binding
con{**of** ty}{=con'}	simple
exb1 **and** 2 **and** exbn	multiple
evb ::=	event binding
con{**of** ty}{**with** ty'}	simple
evb1 **and** 2 **and** evbn	multiple

expressions

const ::=	special constant
INT	integer constant
MP	multiple precision integer constant
REAL	real constant
STRING	string constant
aexp ::=	atomic expression
const	special constant
stop	null behaviour
{**op**} {path}var	variable
{**op**} {path}con	constructor
op {path}=	equality function, (10)
op {path}**orelse**	orelse function, (10)
op {path}**andalso**	andalso function, (10)
op {path}comp	composition operator, (11)
{path}()	0-tuple
(exp1, 2 , expn)	n-tuple
(exp1; 1 ; expn)	sequence
[exp1, 0 , expn]	list

<table>
<tr><td>#[exp1, __0__ , expn]</td><td></td><td>vector</td></tr>
<tr><td>[|exp1, __0__ , expn|]</td><td></td><td>array</td></tr>
<tr><td>let dec in exp1; __1__ ; expn end</td><td></td><td>local declaration</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>exp ::=</td><td></td><td>expression</td></tr>
<tr><td>aexp</td><td></td><td>atomic</td></tr>
<tr><td>exp aexp</td><td>L</td><td>application</td></tr>
<tr><td>exp ren</td><td></td><td>relabelling/aliasing</td></tr>
<tr><td>{do exp} => exp</td><td></td><td>commitment action</td></tr>
<tr><td>signal exp {with exp'} => exp</td><td></td><td>signal action</td></tr>
<tr><td>lab{#aexp}! {exp} => exp</td><td></td><td>output action</td></tr>
<tr><td>lab{#aexp}? {pat} => exp</td><td></td><td>input action</td></tr>
<tr><td>exp id exp'</td><td>L/R</td><td>infixed application</td></tr>
<tr><td>exp -> exp</td><td></td><td>guarded expression</td></tr>
<tr><td>exp : ty</td><td>L</td><td>constraint</td></tr>
<tr><td>exp = exp'</td><td>R</td><td>equality</td></tr>
<tr><td>exp andalso exp'</td><td>L</td><td>conjunction</td></tr>
<tr><td>exp orelse exp'</td><td>L</td><td>disjunction</td></tr>
<tr><td>exp comp exp'</td><td>R</td><td>compositions</td></tr>
<tr><td>exp handle match</td><td>R</td><td>handle exception</td></tr>
<tr><td>exp catch smatch</td><td>R</td><td>catch event</td></tr>
<tr><td>raise exp</td><td></td><td>raise exception</td></tr>
<tr><td>if exp then exp1 else exp2</td><td></td><td>conditional</td></tr>
<tr><td>while exp do exp'</td><td></td><td>iteration</td></tr>
<tr><td>case exp of match</td><td></td><td>case expression</td></tr>
<tr><td>fn match</td><td></td><td>function</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>path ::=</td><td></td><td>structure paths</td></tr>
<tr><td>str "."</td><td></td><td>structure qualifier</td></tr>
<tr><td>path "." str "."</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>match ::=</td><td></td><td>exp match</td></tr>
<tr><td>rule1 | __1__ | rulen</td><td></td><td></td></tr>
<tr><td>rule ::=</td><td></td><td>exp rule</td></tr>
<tr><td>pat => exp</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>smatch ::=</td><td></td><td>signal match</td></tr>
<tr><td>srule1 || __1__ || srulen</td><td></td><td></td></tr>
<tr><td>srule ::=</td><td></td><td>signal rule</td></tr>
<tr><td>exp => exp</td><td></td><td>abbreviated</td></tr>
<tr><td>exp with match</td><td></td><td>with message</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>comp ::=</td><td></td><td>composition operator</td></tr>
<tr><td><code>/\ //\ /\\ //\\</code></td><td></td><td>parallel compositions</td></tr>
<tr><td><code>\/ \\/ \// \\//</code></td><td></td><td>ambiguity compositions</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>ren ::=</td><td></td><td>relabelling/aliasing</td></tr>
<tr><td>"{"hh1, __1__ , hhm"}"</td><td></td><td></td></tr>
<tr><td>"{"rh1, __1__ , rhm"}"</td><td></td><td></td></tr>
<tr><td>"{"ah1, __1__ , ahm"}"</td><td></td><td></td></tr>
<tr><td>ah ::=</td><td></td><td>aliasing</td></tr>
<tr><td>lab11#exp1 __1__ lab1n#expn / lab21 __1__ lab2n</td><td></td><td>(13)</td></tr>
<tr><td>rh ::=</td><td></td><td>renaming</td></tr>
<tr><td>lab11 __1__ lab1n / lab21 __1__ lab2n</td><td></td><td></td></tr>
</table>

hh ::=		hiding
/ lab1__1__labn		
str ::=		structure variable
STR		
var ::=		value variable
VAR		
con ::=		value constructor
CON		
lab ::=		communication label
LAB		
id ::=		infix identifier
VAR		

patterns

apat ::=		atomic pattern
_		wildcard
{op}var		variable
{path}con		constant construction
const		special constant
(pat1, __0__ , patn)		tuple
[pat1, __0__ , patn]		list
pat ::=		pattern
apat		atomic
{op} {path}con apat	L	construction
pat con pat’	L/R	infixed construction
pat : ty	L	constraint
{op}var{:ty} **as** pat	R	layered

types

ty ::=		type expression
tyvar		type variable
{path}tycon		0-ary type construction
ty {path}tycon		1-ary type construction
(ty1; __1__ ; tyn) {path}tycon		n-ary type construction
ty1 * __2__ * tyn		tuple type
ty1 -> ty2	R	function type
bty		behaviour type
(ty)		
bty ::=		behaviour type
{"{"lab1{.ty11}{:ty12}, __1__ , labn{.tyn1}{:tyn2}"}"}btyvar		
tyvar ::=		type variables (all)
TYVAR		
BTYVAR		
tycon ::=		type constructors
TYCON		
btyvar ::=		behaviour type variables

BTYVAR

notes

(1) dec not itself of the form dec; dec

(2) lcs user interfaces use commands that cannot be considered declarations. All command names are reserved words;

(3) See the *User Interfaces* manual for available commands.

(4) Additional syntactic restrictions not expressed here (see recursion section in *Language Reference* manual).

(5) If var has infix status, then at the start of any clause, "op var (apat, apat') ___" may be written " (apat var apat') ___".

(7) "**lazy** id apat1 _0_ apatn = e" is equivalent to "**val rec** id = **fn** apat1 => ... **fn** => apatn = e : 'a lift".

(8) "**agent** id apat1 _0_ apatn = e" is equivalent to "**val rec** id = **fn** apat1 => ... **fn** => apatn = e : _a".

(9) At top-level only, equivalent to "**val** it = exp";

(10) The reserved words =, **andalso**, **orelse** may be used as identifiers in this context, **andalso** and **orelse** become strict functions then.

(11) The composition operators may be used as identifiers in this context, but become strict then functions then.

(12) Except within expressions; structures can't be declared local to expressions.

(13) The richer form "lab11#exp11 _1_ lab1n#exp1n / lab21#exp21 _1_ lab2n#exp2n" is not currently sup ported.