LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Changes to the Standard ML Core Language

by

Robin Milner

LFCS Report Series

ECS-LFCS-87-33
(also published as CSR-240-87)

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

August 1987

Changes to the Standard ML Core Language

Robin Milner

11 Feb 1987

1 Introduction

A few changes to the Core Language of Standard ML are detailed in this report. They were decided in consultation among the following people:

> Dave Berry, Rod Burstall, Luca Cardelli, Guy Cousineau, Mike Gordon, Bob Harper, Gerard Huet, Gilles Kahn, Dave Matthews, Dave MacQueen, Robin Milner, Kevin Mitchell, Brian Monahan, Alan Mycroft, Larry Paulson, Nick Rothwell, Don Sannella, Dave Schmidt, John Scott, Bernard Sufrin, Mads Tofte, Phil Wadler, Ake Wikstrom.

It is intended that from now on all changes to the Core will be upwards compatible; that is, any program written in the Core language as defined in [1], modified by the present changes, will always be valid.[1] The syntax and semantics of the full language will be presented formally in a forthcoming report [2]. The latter report incorporates the present Core changes, and also some changes to Modules (the upper level of the language). While the Modules design is now considered to be stable, more experience of use is needed before we can declare for Modules – as we do for the Core – the intention that all future changes will be upwards compatible.

The present Core changes demand only very slight modifications to existing programs. Precisely, the only modifications demanded are as follows:

- If you have used numeric record labels `#1 #2 ...` , you should now omit the sharp sign and write `1 2 ...` instead.

[1] There is one exception to this policy, at present under discussion. Dave MacQueen has proposed a change to the exception facility which not only adds power but simplifies the language – in particular, it unites the notions of *handler* and *match*. This simplification is so significant that it deserves consideration even though there is a slight violation of the principle of upwards compatible change. If the design group agree, and the semantics is seen to raise no problem, then the change will be adopted; it will also be made clear exactly what change is required to existing programs (indeed, a translation program can be provided to make the changes automatically).

- In the unlikely event that you have used a type variable beginning with two primes, for example `''a` , you should use a type variable beginning with a single prime instead (since those starting with two primes will have a special meaning described below).

2 The Changes

The following changes are made to the Core Language of Standard ML, from April 1987.

2.1 Numeric Record Labels

The numeric record labels, hitherto `#1 #2 ...` , are now to be written `1 2 ...` , without the prefixed sharp sign. The restrictions on use of numeric labels are removed; they may be mixed with non-numeric record labels in record types, patterns and expressions, and are not required to be an initial subsequence of the integers. Thus any finite set of labels is now admissible in a record type. Note that the tuple type $ty_1 * \cdots * ty_n$ is still the type $\{1 : ty_1, \cdots, \overline{n} : ty_n\}$, for $n \geq 2$ ($\overline{n}$ means the numeral which stands for the number n).

2.2 Selector Functions

For each record label *lab* (numeric or otherwise), the selector function *#lab* is introduced; it is a derived form, whose equivalent form is

```
fn { lab = x, ... } => x
```

Thus – just as for the equivalent form – the type of each occurrence of *#lab* must be deducible from the context; sometimes an explicit type constraint may be required for this purpose.

2.3 Abbreviations in Datatype Bindings

A new form of datatype binding is introduced, with the form

db `withtype` *tb* (*)

A simple example of a datatype declaration using this form is

```
datatype ('a,'b)tree = tip of 'a
                     | node of 'b * ('a,'b)forest
    withtype ('a,'b)forest = ('a,'b)tree list
```

The type abbreviations introduced by `withtype` may be used both in the datatype binding and later; thus, the above example is equivalent to

```
datatype ('a,'b)tree = tip of 'a
                     | node of 'b * ('a,'b)tree list;
type ('a,'b)forest = ('a,'b)tree list
```

In the new form (*), *tb* is not interpreted recursively; that is, if *tb* binds a type constructor *tycon*, uses of *tycon* in a right-hand side within *tb* will not refer the present binding but to an earlier one. On the other hand *db* is (as usual) interpreted recursively; the effect of the new form is to allow the abbreviations in *tb* to mediate the recursion of *db*.

Datatype bindings can only occur following `datatype` or `abstype` , so we can define the new form by giving equivalent forms for the two derived forms

`datatype` *db* `withtype` *tb*

`abstype` *db* `withtype` *tb* `with` *dec* `end`

Their equivalent forms are as follows:

`datatype` *db'* ; `type` *tb*

`abstype` *db'* `with type` *tb* ; *dec* `end`

where *db'* is obtained from *db* by simultaneously expanding out the abbreviations defined by *tb*.

A restriction is that `withtype` may appear at most once in any datatype binding. The new form also inherits the general restriction 10(1) in [1] that no binding may bind the same identifier twice.

2.4 Polymorphic Equality

A sub-class of the variables, called *equality* type variables or *eqtype* variables, is now distinguished. They begin with two primes; thus `''a` is an eqtype variable, and `'a` an ordinary type variable. The point of eqtype variables is that they range over equality types as defined in [1], so that now the basic equality and inequality predicates, `=` and `<>` , have the polymorphic type `''a * ''a -> bool` . As a precise explanation, the remainder of this subsection should be taken to replace Section 7.2 of [1] :

An important subclass of types consists of the class of *equality types*, or *eqtypes*, which we now define. In giving the definition we assume that all type abbreviations have been replaced by their unabbreviated forms.

First we determine the type constructors which *admit equality*, which we shall also call *eqtype constructors*, as follows:

- The standard type constructors

  `unit bool int real string list ref`

  and the record-type constructors are all eqtype constructors.

- If datatype constructors $tycon_1, \ldots, tycon_n$ are simultaneously declared in a datatype binding, then they are eqtype constructors provided that every type ty which occurs as a right-hand side in this binding is built from arbitrary reference types and type variables using only eqtype constructors and $tycon_1, \ldots, tycon_n$ themselves.

Thus the only reason that a type constructor may not admit equality is if, in its declaration or in the declaration of any type constructor on which it depends, the function-type constructor `->` is used without being enclosed in the reference-type constructor `ref` . However, an eqtype constructor declared by `abstype` only admits equality in the `with` part of its declaration. We denote an eqtype constructor by *etycon*.

To allow polymorphic eqtypes, we introduce, as a subclass of the type variables, the class of *eqtype variables*, represented by identifiers beginning with two or more primes, for example `''a` , `''b` . We let *etyvar* stand for an eqtype variable. Eqtype variables may only appear in explicit type constraints or in specifications, not in type or datatype bindings. (They will also occur as the result of type-checking.). Their significance is that they may only be instantiated to eqtypes.

We shall let *ety* range over equality types. Their syntax is as follows, assuming that all type abbreviations have been expanded out:

ety ::=		
	etyvar	eqtype variable
	$\langle ety\_seq \rangle etycon$	eqtype construction
	ty `ref`	arbitrary reference-type
	$\{lab_1 : ety_1, \cdots, lab_n : ety_n\}$	record-type, $n \geq 0$

The infixed equality and inequality predicates `=` and `<>` have polymorphic type `''a * ''a -> bool` .

2.5 Polymorphic Assignment

The infixed assignment function `:=` is now given the polymorphic type `'a ref * 'a -> unit` . This extension has been shown to be sound by Mads Tofte. However, any use of the reference function `ref` in an expression must still be with a monomorphic type *mty* `->` *mty* `ref` in expressions (though it may be used polymorphically in patterns, with type `'a -> 'a ref`). It is hoped to find a robust and easily understood relaxation of this restriction in the near future.

2.6 Polymorphic Exceptions

It is necessary to introduce a restriction on the use of polymorphic exceptions, due to an unsoundness recently discovered by MacQueen. The restriction is as follows:

> If an exception *exn* is declared with a polymorphic type, or if such a type is inferred for it by the type checker, then no handler of *exn* may occur within a function expression which lies within the scope of the declaration of *exn* .

Thus the following is forbidden:

> `let exception` *exn*`:'a in` ··· `(fn` ··· `handle` *exn* ··· `)` ··· `end`

Note that there is no restriction on occurrences of `raise` expressions, nor upon handlers of monomorphic exceptions.

References:

1. R.Harper, D.MacQueen and R.Milner, Standard ML. Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, CS Department, Edinburgh University, 1986.

2. R.Harper, R.Milner and M.Tofte, The Semantics of Standard ML. Forthcoming ECS-LFCS Report, as above, 1987.