Unifying Exceptions with Constructors in Standard ML

Andrew Appel, David MacQueen, Robin Milner, Mads Tofte

May 30, 1988

1 Introduction

The Standard ML Core Language, as described in [1], was modified by several changes detailed in [2]. It was declared in [2] that, with one exception, all later changes to the language will be upwards compatible — i.e. any program written in the language defined by [1] and amended by [2] will be always be valid. The exception in fact concerns a simplification of the exception facility. It was declared there that this simplification would be adopted, provided that the design group agreed and that careful semantic analysis raised no difficulties. Both these conditions have been met, and the change will now be adopted.

"The Semantics of Standard ML, Version 1" [3] applies to the language of [1] modified by [2], but does not incorporate this new change (though it mentions it as under consideration). Version 2 of [3] will incorporate it.

Existing programs will be syntactically incorrect in the changed language, but it is easy to convert them mechanically to the new syntax so that they will run exactly as before. Present implementations can usefully provide this translation facility, at least for a limited period.

In Section 2 below we describe the new change informally. In Section 3 we describe precisely the changes to syntax and to predeclared objects (i.e. the initial basis).

2 Informal description of the change

It was always understood that the semantics of matching exceptions is very much like the semantics of matching ordinary constructors, but when the current design of Standard ML was established we could not see how to unify the two concepts. We therefore accepted two parallel but similar constructs, requiring special syntax for exception matching with the additional keywords `||` and `?`. Besides requiring extra syntax, this approach also leaves a serious gap in functionality; it does not allow a handler to re-raise an exception (without knowing its name) having failed to match it.

In the light of further design experience and thorough semantic analysis it appears that the unification of exceptions and data constructors causes no semantic difficulty and little inconvenience. Furthermore it yields greater simplicity of design and considerable gain in expressive power. Nor are there any implementation problems; the new scheme has been successfully used for the internal representation and processing of exceptions in the Bell compiler of MacQueen and Appel.

One corollary of this change is that exception names, being now data constructors, can no longer be distinguished from variables by syntactic context. We therefore abandon the "homonym convention" whereby an exception may bear the same name as the function which raises it (e.g. the standard exception `hd` is raised by the standard function `hd`). Instead, for standard exceptions, we adopt a convention that exception constructors begin with a capital letter, so that `hd` will now raise `Hd` when applied to the empty list. Users may, but need not, choose to maintain the convention. For exceptions raised by standard functions with symbolic identifier names, like "`+`", we adopt names like `Sum`.

We now give a simple example of the modification required to programs. Consider the following ML program, in the unmodified language:

```
exception found : string
exception disaster
  . . .
val j = ( . . . ) handle found with "money" => 10
                                  | "old boots" => 11
                                  | _ => 12
                      || disaster => -10
                      || io_failure => -11
                      || hd => -12
                      || ? => 0
```

The new version will be as follows:

```
exception found of string
exception disaster
  . . .
val j = ( . . . ) handle found "money" => 10
                       | found "old boots" => 11
                       | found(_) => 12
                       | disaster => -10
                       | Io(_) => -11
                       | Hd => -12
                       | (_) => 0
```

Note a few points:

- In the old system the clause "`io_failure => -11`" is an abbreviation for "`io_failure with (_) => -11`", but there is no corresponding abbreviation in the new scheme (it would violate the normal treatment of patterns).

- It seems more elegant to adopt `Io` instead of `io_failure` as the exception constructor for I/O failures.

- The declaration "`exception disaster`" is not treated the same as the declaration "`exception disaster of unit`" (in contrast with the old scheme). With the latter definition, as for constructors in general, the correct clause in the match following handle would be "`disaster(_) => -10`".

Under the new scheme, an exception name like `found` represents a particular kind of data constructor, and what follows `handle` — previously a handler — is just an ordinary match. The new scheme requires the existence of pre-declared (standard) datatype `exn`, and the declaration

```
exception found of string
```

declares a new constructor whose type is `string->exn`. The new datatype `exn` is special in just two ways:

- New constructors may be declared for it, as above;

- Values of type `exn` (and no other) may be raised by `raise` and handled by `handle`.

In all other respects `exn` is an ordinary datatype; its values are first-class objects and may be manipulated just like any other values. In particular, they may be constructed outside the context of a `raise` expression, and matched outside of a `handle` expression. Continuing the above example:

```
val e = found "money"

fun my_handler(found "money") = false
  | my_handler(f) = raise f

(if a<b then raise e else true)
    handle ex => my_handler ex
```

The next example shows the added benefit of re-raising exceptions:

```
(increment x; f(); decrement x)
   handle e => (decrement x; raise e)
```

The expression above is an "unwind-protect" (in the language of LISP); it ensures that if an exception is raised from `f()`, then the variable `x` will be restored to its original value, and the exception will be propagated upwards.

The new type `exn`, together with the ability to define any number of constructors which "wrap up" values of any type as exceptions, and the ability to "unwrap" these values by ordinary pattern-matching, provides in effect a truly *general* type in ML (in the sense used by Strachey). This is totally independent of the facility which allows us to raise and handle these wrapped-up objects or packets. Users will therefore sometimes use the type in its*general* capacity only, and will in that case find the terminology `exception` and `exn` (for keyword and type respectively) a little misleading. We have nevertheless decided to keep this terminology, since the situation can be improved in the future by one of two changes, each of which will be upwards compatible:

- We may allow the user to declare many types all of which admit extension by adding new constructors. Then the type `exn` will be just one such type which is standard; the `exception` declaration form will be a derived form of the general form of "new constructor" declaration.

- Or we may conclude that only one such type is needed. In that case we can call it (say) `general`; we can have `exn` as an alias for it; and we can introduce a less colourful keyword like `newcons` for introducing new constructors, allowing `exception` as a synonym for use at the user's discretion.

Any change to terminology now could preempt this choice.

3 Details of the change

3.1 Identifier classes

The identifier class of exception names (*exn*) will now be called exception constructors (*excon*). An identifier is an exception constructor if it is within the scope of a declaration which introduced it as such. As for ordinary value-constructors, no value declaration can make a "hole" in this scope, since any occurrence of either kind of constructor in a pattern is interpreted as that constructor, not as the binding occurrence of a new variable.

3.2 Reserved words

The reserved words `||` and `?` are abolished. The keyword `with` remains for use with `abstype`, but loses its role in handlers.

3.3 Syntax classes

The syntax classes *handler* and *hrule* are abolished. The classes *eb* (exception bindings), *aexp* (atomic expressions), *exp* (expressions), *apat* (atomic patterns) and *pat* (patterns) are changed as follows, using the conventions of [1], Table 4, p32:

		ADDED	REMOVED
eb	::=	*excon* << `of` *ty* >>	*exn* <<: *ty* >><<= *exn*′ >>
		excon = *excon*′	
aexp	::=	<< `op` >> *excon*	
exp	::=	`raise` *exp*	`raise` *exn* << `with` *exp* >>
		exp `handle` *match*	*exp* `handle` *handler*
apat	::=	*excon*	
pat	::=	<< `op` >> *excon apat*	
		pat excon pat	

The typing rules are:

- The type in an exception binding may contain no type variables. (*Note*: this may be relaxed at the same time as polymorphic references are introduced.)
- The type of the *exp* following `raise` must be `exn`.
- The type of the *match* following `handle` must be of the form *ty*`->exn`.

The following notes will give an understanding of the semantics; a formal definition will appear in Version 2 of [3].

1. At each evaluation of the exception binding *excon* `of` *ty*, a new constructor of type *ty*`->exn` is generated and bound to *excon*. This agrees with the old scheme. It is necessary to ensure that accidental coincidence of names does not cause unintentional handling of an exception which has been propagated by `raise` outside the scope of the constructor which built it. The typeless exception binding *excon* similarly declares a new constant of type `exn`. The binding *excon* = *excon*′ declares a new identifier for the constructor (bound to) *excon*′.

2. The *match* following `handle` treats unmatched patterns differently from ordinary matches. Instead of an implied rule `(_) => raise Match` after the match, there is an implied rule `e => raise e`, so that unmatched exceptions are not handled by this handler, but are propagated upwards.

3. The two declarations `exception e of unit` and `exception e` are not the same, even though the corresponding declarations in the old scheme were the same. Just as with ordinary constructors, carrying a value of type `unit` is not the same as carrying no value. See the example of `disaster` in Section 2 above.

3.4 Standard type constructors

There is a new standard type constant (nullary constructor) `exn`, the type of exceptions. It does not admit equality.

3.5 Standard exception constructors

Standard constants of type `exn` are introduced as follows:

```
Match  Bind  Interrupt  Ord  Chr
 Div  Mod  Floor  Sqrt  Exp  Ln
```

in place of the existing corresponding standard exception names. The existing exception names `* / + -` (at present homonymous with the functions which raise them) become standard constants of type `exn` as follows:

```
Prod  Quot  Sum  Diff
```

Finally, there is an exception constructor

```
Io
```

of type `string->exn`, in place of the exception name `io_failure`.

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.Milner, *Changes to the Standard ML Core Language*, Report ECS-LFCS-87-33, Laboratory for Foundations of Computer Science, CS Department, Edinburgh University, 1987.

3. R.Harper, R.Milner and M.Tofte, *The Semantics of Standard ML, Version 1*, Report ECS-LFCS-87-36, Laboratory for Foundations of Computer Science, CS Department, Edinburgh University, 1987.