This file is SI0:[LC]mlchanges.doc

Differences Between VAX and DEC-10 ML

DEC-10 syntax is in quotes "---"; VAX syntax is in twiddles ~----.

Identifiers

An identifier is either a sequence of letters,numbers and primes starting with a letter, or a sequence of special characters. The special characters are grouped into three classes:

- 0: !$^¦&
- A: -<+:=#>
- B: *?~@_

In forming an identifier, every adjacent pair of special character must 'stick'; two special chars stick if they belong to the same class or if at least one is of class 0, i.e. they will not stick only if one is of class A and the other one of class B. Moreover the chars of class 0 stick to the inner side of parenthesis (),[],{}.

List Operators

The infix operator "." (list cons) is now ~_~.
The infix operator "@" (list append) is now ~::~.

Tokens and Token Lists

Token list quotes "``" are now double quotes ~"~. The following characters have special meanings in token lists (but not in tokens):
~ ~ (blank): separates tokens; ~_~ (underscore): blank (replaces "/S");
~/\~ (escape backslash): empty token (e.g.: "/\ /\" = [``;``]).
Tokens and token list are printed at the top level in the same form in which they are read, that is with all the "/" and with no funny character inside. The same apply to ~printtok~ but not to ~outtok~.

Primitive functions on tokens are ~explode: tok -> tok list~ and ~implode: tok list -> tok~. Two other primitives have been added for int-to-ascii conversions: ~explodeascii: tok -> int list~ and ~implodeascii: int list -> tok~.

The semantic limit on the length of tokens is 64K chars.

The syntactic limit on the length of token quotations is also 64K chars. There is no syntactic limit on the length of token list quotations provided that each token meets its maximum length.

However there is a general limitation on the length of a line of text which implies that a token or token list quotation should be broken by a carriage return at least every 256 characters.

Numbers

Numbers are integers in the range -32767..32768.

" -3 " is now - -3 -; it is then possible to distinguish -f(-)- from -f(-)-.

Equality

It is unfortunately impossible to use the mathematical definition of equality, as we want a decidable equality predicate. Hence equality is considered to be a predefined overloaded operator: an infinite collection of decidable equality operators, whose types are instances of -(* # *) -> bool-.
This definition still gives a great deal of freedom in the choise of the semantics of -=- because we can overload it to the any arbitrary degree.
In the current implementation, equality only works on monotypes which are not function spaces or isomorphic domains; the rationale for these restrictions is:

- If only monotypes are involved, then the compiler can produce code which does not need to perform run-time type checking (consider the polymorphic equality -\a.a=a-: what code should we produce for it?). Note that not even monomorphic equality can be compiled as a subroutine; the compiler must produce specialized in-line code for every occurrence of -=- or -<>- (inequality).
- Equality on functions is undecidable.
- Equality over isomorphic domains might be implemented as equality over the corresponding concrete representations.
  However in this case we would have troubles in the use of isomorphisms as abstract types; for example equality would not be transparent to a changes of representation (e.g. from a int list to a int->int).
  Also, if we implement abstract sets by concrete multisets, equality could distinguish between sets which should be equal, giving dangerous insights on the chosen representation.

Here are examples of right and wrong comparisons (at the top level):

Right	Wrong
`() = ()`	
`true = false`	
`3 = 3`	
`` `a` = `` ``	
`(2,`a`) = (2,`a`)`	
`[] = ([]: int list)`	`[] = []`
`[1;2] = [3]`	
`inl 3 = (inl 3: int+.)`	`inl 3 = inl 3`
`inl 3 = inr ()`	
`\a,b. a=b : tok#tok -> bool`	`\a,b. a=b`
	`I = K`
	`(absset[3]) = (absset[3])`

Some of the wrong comparisons might be trivially solved at compile-time, but where do we stop?

New semantics of declarations

"letrec ---" is now -let rec ---; this gives extra flexibility in

declarations, like in:
-let a=3 and rec f n = ...f(n-1)... in ---~.

The new set of environment operators is:

-b = t~	is just like "b = t" where -b~ is a binder and -t~ is a term. It exports a single declaration for -b~.
-A and B~	is just like "A and B": it exports the declarations of -A~ and -B~, but the declarations of -A~ are not exported into -B~ and vice versa.
-rec A~	the declarations of -A~ are exported into -A~ and also outside -rec A~.
-A enc B~	(enclose) the declarations of -A~ are exported into -B~ (but not vice versa) and -A enc B~ exports the declarations of -B~ and the declarations of -A~ which are not redefined in -B~. For example -let A enc B in C~ is equivalent to -let A in let B in C~ (but consider -{A enc B} and {C enc D}~).
-A ins B~	(inside) the declarations of -A~ are exported into -B~ (and not vice versa) but -A ins B~ only exports the declarations of -B~. This means that -A~ defines a set of own variables for -B~.
-A with B~	behaves like -enc~ on types and like -ins~ on values; useful to define abstract types where the abstract type itself is exported but the isomorphism is hidden.
-A ext B~	(extend) is just -B enc A~.
-A own B~	is just -B ins A~.
-type A~	declares a new set of types. Note that -A~ can contain all the environment operators except -type~ and -with~. Mixing type and value declarations is useful in defining abstract types.
-b <=> t~	(only within a -type~ declaration) creates an isomorphism between the type binder -b~ and the type term -t~. It exports a type identifier -b~ and two function identifiers -absb~ and -repb~. Together with -ins~ or -with~ can produce abstract types.

Notes:

It is now possible to decide whether or not to export the abstract type together with its operators: the program

```
-let type b <=> t
 ins op1=... and opn=...~
```

does not export the type -b~ which is hidden by -ins~, while

```
-let type b <=> t                -let type b <=> t
 ins {type b = b}          ==     with op1=...
 and op1=... and opn=...~         and  opn=...~
```

exports abstract type -b~ (by a defined type -b~ or directly).

It isn't allowed to use -ins~ (or -with~) within a -rec~, like in -rec {... ins ...}~. This can be justified as follows: -let A ins b=t~ is equivalent to -let b=let A in t~; hence -let rec {A ins b=t}~ is -let rec {b=let A in t}~ which is not allowed in ML as recursive declarations must have \-terms on the right hand side. Allowing -rec ins~ also introduces extra complexity in implementations without any apparent gain because we can always rephrase -rec (A ins B)~ as -A ins rec B~ (A would not be recursive in any case).

It is not allowed to use ~=~ within a ~rec type~ declaration: ~<=>~ must be used instead (note however that ~<=>~ can be used in non recursive declarations). A semantic justification is that recursive domains work only up to isomorphism. From a pragmatic point of view ~=~ in a ~rec type~ forces the typechecker to work on circular structures, which is not appealing.

New syntax of declarations

This is a semantics-preserving translation table:

"lettype ---"	==>	~let type ---~
"letrec ---"	==>	~let rec ---~
"abstype B1 = T1 and ... and Bn = Tn with ---"	==>	~let type B1 <=> T1 and ... and Bn <=> Tn with ---~
"absrectype B1 = T1 and ... and Bn = Tn with ---"	==>	~let rec type B1 <=> T1 and ... and Bn <=> Tn with ---~

Similar translations apply to "where".

The syntax of declarations is now:

```
Decl ::=
  (Bind ¦ FunBind [":" Type]) "=" Term ¦
  Decl ("and" ¦ "enc" ¦ "ext" ¦ "ins" ¦ "own" ¦ "with") Decl ¦
  "rec" Decl ¦
  "type" TypeDecl ¦
  "{" Decl "}".

TypeDecl ::=
  TypeBind ("=" ¦ "<=>") Type ¦
  TypeDecl ("and" ¦ "enc" ¦ "ext" ¦ "ins" ¦ "own" ) TypeDecl ¦
  "rec" TypeDecl ¦
  "{" TypeDecl "}".
```

where the binding power of operators is:

(and) > (rec = type) > (enc = ext = ins = own = with)

and the infix ones are right associative. Ex:

~let rec f a = ... and g b = ... ins x = ... and y = ...~ means:

~let {{rec {f a = ... and {g b = ...}} ins {x = ... and {y = ...}}}~

Restrictions:

```
    -rec {... ins ...}-    is illegal (no -ins- (or -with-) within a -rec-)
    -rec type ... = ...-   is illegal (you must use -<=>- within
                                       a -rec type- or -type rec-)
```

Iteration

The iterative forms of failure ("!" and "!!") have been abolished.
The iterative conditional (if-then-loop) will be reintroduced soon.

References

There is a type -* ref- with operations

```
   -ref : * -> * ref-          (create a new reference)
   -@   : * ref -> *-          (dereferencing)
   -:=  : * ref # * -> .-      (assignment)
```

The "letref" feature is not supported; "letref a = 3 in a:=a+1"
must now be done by -let a = ref 3 in a:=@a+1-.
Note also that there is no varstruct facility on the left of -:=-.
On the other hand, some restrictions on the use of "letref"
do not apply to -ref-; for example it is possible to write
at the top level -let a = ref[];- while "letref a = [] : int list;"
must specify a monotype for "[]".
The new references can be inserted in data structures at any level.
Arrays (with linear access time) can be defined as:

```
  -let type * array <=> * ref list-.
```

Labelled Types

Labelled products and sums have been introduced; they model respectively
records and variants.

- RECORDS.

A record is a collection of unordered named fields:

```
  (¦a1=t1; ... ; an=tn¦)     n>=0
```

where `ai` are fields selectors (identifiers) and `ti` are values (terms).
The type of a record is a labelled unordered product:

```
  (¦a1=t1; ... ; an=tn¦) : (¦a1:T1; ... ; an:Tn¦)   where ti : Ti
```

A record field can be selected by its field name;

```
  (¦a1=t1; ... ; an=tn¦).ai = ti
```

Application of a function to a record gives a sort of call-by-keyword
facility, because records are unordered:

```
  (\(¦a=a; b=b¦). a,b) (¦b=2; a=1¦)  =  1,2
```

When a record is used in a varstruct, the effect of a WITH statement
is obtained.

- VARIANTS.

A variant type is an unordered disjoint union of types:

```
  [¦a1:T1; ... ; an:Tn¦]    n>=0
```

an object of this type embodies exactly one of the possible alternatives offered by the type:

```
  [¦a1=t1¦] , ... , [¦an=tn¦]     where ti : Ti
```

we can ask whether a variant object `is` one of the alternatives:

```
  ([¦a=t¦] is a) = true       ([¦a=t¦] is b) = false
```

and then extract its content:

```
  ([¦a=t¦] as a) = t          ([¦a=t¦] as b)   fails
```

Variants can be used in varstructs, for example:

```
  (\[¦a=a¦]. a)  =  (\x. x as a)
```

A CASE statement is provided:

```
  case x of [¦a1=v1. t1; ... ; an=vn. tn¦]
```

where v1...vn are varstructs; for example:

```
  case t: [¦Ide: tok;
            Lamb: (¦Bind: tok; Body: Term¦);
            Appl: (¦Fun: Term; Arg: Term¦) ¦]
  of [¦Ide=name. outtok name;
       Lamb=(¦Bind=Bind; Body=Body¦).
         (outtok`(\`; outtok Bind; outtok`. `; Print Body; outtok`)`);
       Appl=(¦Fun=Fun; Arg=Arg¦).
         (outtok`(`; Print Fun; outtok` `; Print Arg; outtok`)`) ¦]
```

- ABBREVIATIONS.

In terms and varstructs:

```
  (¦a¦)    =    (¦a=a¦)
  [¦a¦]    =    [¦a=()¦]
```

In types:

```
  (¦a¦)    =    (¦a:a¦)
  [¦a¦]    =    [¦a:.¦]
```

In case statements:

```
  [¦a. t¦] =    [¦a=(). t¦]
```

Note that the abbreviations for variants are intended to model enumerated types:

```
  let type color = [¦red; green; blue¦];      [¦red¦] : color
```

- RESTRICTIONS.

It is not allowed to write something like -\x. x.a- unless the type of -x- is further specified by the context; the typechecker must be able to infer exactly the name and type of all the fields

of every record-object (i.e. every record-object must have a RIGID type, as explained below). This restriction is not due to typecheck problems but only to efficiency considerations. This restriction allows the compiler to generate efficient code for field selection (i.e. direct access, as opposed to associative search).

It is not allowed to write something like -[¦blue¦]- unless the type of -[¦blue¦]- is further specified by the context; the typechecker must be able to infer exactly the name and type of all the variants contained in the type of every variant-object (i.e. every variant-object must have a RIGID type, as explained below). This restriction too is only due to efficiency considerations so that the compiler can generate efficient code for the case statement (i.e. a jump table as opposed to a sequence of if-then-else).

- TYPECHECKING.
Partially specified records and variants are called FLEXIBLE, while fully specified ones are RIGID, in much the same way as we might call (a+b) a rigid binary dijoint sum and (a+b+*) a flexible binary dijoint sum. There is no way to express syntactically this distinction (as an attempt to keep things simple) but it is sometimes useful to know the default actions taken by the typechecker. The point is that two rigid types will typecheck only if they have the same named fields, while they will typecheck to the 'union' of their fields if both of them are flexible. If only one is flexible, the fields of the flexible type must be included in the fields of the rigid one.

Here are the rules: a record constant has a rigid type; a variant constant has a flexible type; -x.a-, -x is a- and -x as a- assign a flexible type to -x-; type definitions -let type r=(¦...¦) and v=[¦...¦]- and type specifications -r:(¦...¦),v:[¦...¦]- assign rigid types both to -r- and to -v-. The consequences are sometimes intriguing.

Sequencing

Sequencing of terms "e1; ... ;en" requires extra parenthesis -(e1; ... ;en)-.
All the side effecting primitives return now -()- (they were identity functions):

```
:=        :  * ref # * -> .
printdot  :  . -> .
printbool :  bool -> .
printint  :  int -> .
printtok  :  tok -> .
outtok    :  tok -> .
```

thus it is possible to eliminate the extra -()- at the end of sequencings, which where needed in cases like:
"x => a:=true; b:=0; () ¦ a:=false; c:='-'; ()". It is now:
-x => (a:=true; b:=0) ¦ (a:=false; c:='-')-.

Fix operators

Local prefixes,infixes,suffixes and nonfixes can be declared by:

```
-syntax prefix do
        infix  <------>
        suffix square
        nonfix +
 in ... -
```

They only operate in the scope - ... - above and can be used as fixes in declarations. Both identifiers (-abc123'-) and symbols (-~<~>~-) are legal variables, and they can have any fixity.

Type fixes can be independently defined by -syntax type ... in ...-. Independently means that -++-, for example, can be both an infix function and a prefix type operator.

No error is ever given because of misplaced fix operators; when a fix operator is out of place it is taken as a nonfix (Ex: -[+; -; *; /]- ; -,,,- ; -\=,a,b. a=b-).

Garbage Collection

There is a primitive identity function -collect: * -> *-. The message 'Collecting' is printed just before collection.