UNBOXEDOPS

By: Jan Pedersen(Pedersen.PA @ Xerox.com] and Larry Masinter (Masinter.PA @ Xerox.com]

The module UNBOXEDOPS is intended to assist those interested in high-performance, scalar, floating-point arithmetic. The basic trick is to perform floating point arithmetic on the stack, utilizing special, unboxed, floating-point opcodes, an ugly but usually effective solution. This method of eliminating floating-point number boxes is likely to change, but in the interim a combination of compiler declarations and explicit evocations of unboxed operations, as described below, will allow the interested user to eliminate a high percentage of floating-point number boxes. This module and the methods described are "safe", i.e., the declarations won't cause your programs to crash, and if it works with the declarations it will also work without them.

Unboxed floating point tricks help out only 1108's with floating point hardware or 1186's with floating point microcode. Unfortunately, they may make performance even worse on 1108's without floating point hardware, although the performance degradation is probably not too severe.

There exist opcodes which perform floating point arithmetic on the stack (that is, on the bits of those numbers, rather than pointers to those bits). These opcodes are only emitted by the byte compiler if arithmetic occurs in an unboxed context. One example of an unboxed context is arithmetic on a record field defined to be of type FLOATP, another is arithmetic on a variable declared to be of TYPE FLOAT . However, the compiler will box across function boundaries and in a return context. Furthermore, there exist more unboxed opcodes than are used by the compiler (unboxed comparison springs to mind).

UNBOXEDOPS defines macros/functions so that these additional opcodes may be exploited in an unboxed context. These macros/functions include:

UFABS, UFEQP, UFGEQ, UFGREATERP, UFIX, UFLEQ, UFLESSP, UFMAX, UFMIN, UFMINUS, and UFREMAINDER,

which behave identically to there non-U namesakes, except that the operations are done on the stack without generating floating point boxes.

For those unfamiliar with unboxed compiler declarations a short description follows:

Using (DECLARE (TYPE FLOATING x y z)) to reduce number boxes

Consider the silly function:

```
(DEFINEQ (FIE  (N)
        (bind (SETQ X 0.0) (SETQ Y 2.0) for I from 1 to N
              do (SETQ X (FPLUS X (FTIMES Y Y)))
              finally (RETURN X))))

(TIMEALL (FIE 100))
```

returns a CPU time of .025 and reports 200 FLOATP boxes produced. Now, consider

```
(DEFINEQ (FOO  (N)
        (bind (SETQ X 0.0) (SETQ Y 2.0) for I from 1 to N
                declare (TYPE FLOAT X Y)
                do (SETQ X (FPLUS X (FTIMES Y Y)))
                finally (RETURN X))))
 (TIMEALL (FOO 100))
```

returns a CPU time of .003 seconds and reports just one floatp box produced.

Essentially the (TYPE FLOAT X Y) declaration is a promise to the compiler that X and Y will hold FLOATP's , so arithmetic may be done unboxed (that is on the value itself, instead of on a pointer to the value, which is the usual case) if possible. The key issue is what is meant by "if possible".

The compiler is conservative. It will perform unboxed arithmetic only on built-in arithmetic functions (PLUS , TIMES, DIFFERENCE, etc), which have unboxed counter parts, and will otherwise box across function boundaries regardless of compiler declarations.

For example:

```
(DEFINEQ (FOOBAR  (N)
        (bind (SETQ X 0.0) (SETQ Y 2.0) for I from 1 to N
                declare (TYPE FLOAT X Y)
                do (SETQ X (FPLUS X (LOG Y)))
                finally (RETURN X))))
```

then

```
(TIMEALL (FOOBAR 100))
```

returns a CPU time of .049 with 601 FLOATP boxes produced (some of which come from the LOG (five per function call)).

Also, the compiler will box in a return context.For example

```
(DEFINEQ (BAR  (N)
        (bind (SETQ X 0.0) for I from 1 to N
                 declare (TYPE FLOAT X )
                 do (SETQ X
                        (PROG ((Y 2.0))
                                (DECLARE (TYPE FLOAT Y))
                                (RETURN (FTIMES Y Y))))
                finally (RETURN X))))
```

then

```
 (TIMEALL (BAR 100 ))
```

returns a CPU time of .022 with 301 FLOATP boxes produced -- notice that BAR seems like it should behave like FOO.

Indeed that is the the greatest drawback of the unboxed arithmetic as it stands now -- it is not always easy to predict what is going to happen -- there are even traps where indiscriminate uses of TYPE FLOAT declarations will actually produce MORE boxes than without them. This is the case if, for

example, you use comparison operators (GREATERP, etc) since the compiler boxes each operand before invoking them.

The BAR example may be fixed up as follows:

```
(DEFINEQ (BAR  (N)
        (bind (SETQ X 0.0) for I from 1 to N
                declare (TYPE FLOAT X )
                do (SETQ X
                        (PROG ((Y 2.0) RESULT)
                                (DECLARE (TYPE FLOAT Y RESULT))
                                (RETURN (SETQ RESULT (FTIMES Y Y))))
               finally (RETURN X)))
```

then

```
(TIMEALL (BAR 100))
```

returns a CPU time of .008 with 101 FLOATP boxes produced. Note that the compiler still boxes the result returned by the PROG.

The best way to find out what is happening is to use a combination of TIMEALL and INSPECTCODE . Unanticipated boxing behavior will show up as BOX opcodes -- if you find a sequence of opcodes UNBOX , BOX , function call, UNBOX , then you know you are in trouble. TIMEALL will report the total number of boxes produced.

Basically TYPE FLOAT declarations are best used in tight inner loops of the sort illustrated in FOO.

With all these caveats, I think it is only fair to say that considerable performance inprovements can be realized with judicious use of the TYPE FLOAT declarations; my measurements indicate a factor of ten.

Additional note: TYPE FLOAT vars are by necessity LOCALVARS.

Lyric compatibility note: Allthe entries described for this module are in the Interlisp package. Only the Byte compiler pays attention to TYPE FLOAT declarations -- i.g. use of TYPE FLOAT declarations will be ignored by the XCL compiler.