

The Modula-3 FOR loop

Roland Illig <roland.illig@gmx.de>

September 25, 2005*

Modula-3 is a simple and safe programming language. The Modula-3 FOR statement is defined such that using it with very large or very small integers can result in unexpected overflow errors, which could terminate the program. In this article, a revised definition of the FOR statement is developed from the current definition, which behaves more like the user would expect.

Contents

1	The current FOR loop	1
2	How to write overflow checks	2
3	Implementation costs	3
4	Further optimization	4
5	Differences to the original FOR loop	7
6	Implementation costs	7
7	Conclusion	7

1 The current FOR loop

`FOR var :=first TO last BY step DO statements END`

Section 2.3.16 of [M3def] defines the FOR loop. There is an explicit note in the last paragraph that the FOR loop should not be used when the *last* expression is equal to `LAST(INTEGER)`. This warning is useful, but misses the point that in case *step* < 0, similar problems exist for `FIRST(INTEGER)` too, as well as for values near those extrema when the absolute value of *step* is greater than 1.

*\$Id: modula-3-for-loop.tex,v 1.12 2005/09/25 10:22:49 roland Exp \$

```

VAR
  i := ORD(first); done := ORD(last); delta := step;
BEGIN
  IF delta >= 0 THEN
    WHILE i <= done DO
      WITH id = VAL(i, T) DO S END;
      INC(i, delta)
    END
  ELSE
    WHILE i >= done DO
      WITH id = VAL(i, T) DO S END;
      INC(i, delta)
    END
  END
END
END

```

Figure 1: FOR loop, as defined in [M3def], section 2.3.16

The semantics of the FOR loop are defined in terms of code rewriting. That code can lead to integer overflow if the *last* expression is close to *FIRST(INTEGER)* or *LAST(INTEGER)*. According to the code rewriting given in the language definition, the statement

```
FOR  $i := \text{LAST}(\text{INTEGER}) - 5$  TO  $\text{LAST}(\text{INTEGER}) - 4$  BY 16 DO ... END
```

leads to overflow as follows. The condition $i \leq \text{done}$ holds, the statements are executed with $id = \text{LAST}(\text{INTEGER}) - 5$, and after that, i is incremented by 16, leading to a value outside of the valid *INTEGER* range.

I tested the code with the “Critical Mass Modula-3 version d5.2.7” compiler on the NetBSD2/i386 platform. The generated code does not check for overflow (as documented), but simply uses wrap-around arithmetics, so that the FOR loop leads to an endless loop. This behavior has probably not been intended by the language designers. (I deliberately chose *step* to be a factor of 2^{32} , the number of different *INTEGER* values on my platform. If it hadn’t been a factor of 2^{32} , the loop would have terminated after wrapping around for some times.)

The code from [M3def] is given in figure 1. To wipe out the anomalies, some code must be inserted to check for integer overflow before i is incremented. The result is shown in figure 2.

2 How to write overflow checks

The overflow checks that have been inserted take the commonly used form of overflow checks, which is described here.

```

VAR
  i := ORD(first); done := ORD(last); delta := step;
BEGIN
  IF delta >= 0 THEN
    WHILE i <= done DO
      WITH id = VAL(i, T) DO S END;
      IF i > LAST(INTEGER) - delta THEN EXIT END; (* new *)
      INC(i, delta)
    END
  ELSE
    WHILE i >= done DO
      WITH id = VAL(i, T) DO S END;
      IF i < FIRST(INTEGER) - delta THEN EXIT END; (* new *)
      INC(i, delta)
    END
  END
END
END

```

Figure 2: FOR loop with overflow checks

If the range of *INTEGER* were unlimited, in case of $\delta \geq 0$ one could simply write

```
IF  $i + \delta > \text{LAST}(\text{INTEGER})$  THEN EXIT END
```

But as the range of valid integers is limited, the addition might overflow, so the condition must be transformed into one that can never overflow. This can be reached by subtracting δ on both sides of the condition, leading to the following code.

```
IF  $i > \text{LAST}(\text{INTEGER}) - \delta$  THEN EXIT END
```

As the value of δ is between 0 and $\text{LAST}(\text{INTEGER})$ inclusively, the expression $\text{LAST}(\text{INTEGER}) - \delta$ is in the same range. Integer comparisons cannot lead to overflow, so the code is robust now.

3 Implementation costs

The downside of this solution is that the number of instructions inside the FOR loop has increased, which can make the generated code unacceptably slow. Some optimizations can reduce the amount of the additional code:

- The value of the expression $\text{LAST}(\text{INTEGER}) - \delta$ can be computed once before the loop starts, as the value of δ does not change inside the FOR loop.
- In case $\delta \geq 0$, the overflow check can be omitted completely if the condition $\text{done} \leq \text{LAST}(\text{INTEGER}) - \delta$ is known to be true, as $i \leq \text{done}$ is known to

```

VAR
  i := ORD(first); done := ORD(last); delta := step;
BEGIN
  IF delta >= 0 THEN
    LOOP
      IF i > done THEN EXIT END;
      WITH id = VAL(i, T) DO S END;
      IF i > LAST(INTEGER) - delta THEN EXIT END;
      INC(i, delta)
    END
  ELSE
    LOOP
      IF i < done THEN EXIT END;
      WITH id = VAL(i, T) DO S END;
      IF i < FIRST(INTEGER) - delta THEN EXIT END;
      INC(i, delta)
    END
  END
END
END

```

Figure 3: FOR loop rewritten using LOOP

be true from the condition in the WHILE loop and the \leq relation is transitive. If the *last* expression is an integer constant or an enumeration constant, it is easy to decide whether the check needs to be done or not.

- Similarly, in case $\delta < 0$ the check can be omitted if the condition $done \geq FIRST(INTEGER) + \delta$ is known to be true.

4 Further optimization

The code from figure 2 can be transformed to the equivalent code of figure 3. This code reveals that there is much similarity between the two conditions in the IF statements inside the LOOP. This similarity can be used to further optimize the generated code by combining the two IFs. If the two IF statements were adjacent, this would be simple. But as they are not, there's more work to do.

For the rest of the transformations only the branch for $\delta \geq 0$ will be considered in detail, as the steps are very similar in the other branch.

The check for $i > done$ can be split and moved. The first instance goes out of the LOOP. As the EXIT statement would be invalid here, the condition is negated. The second instance is moved after the INC statement. The resulting code is:

```

IF i <= done THEN

```

```

LOOP
  WITH id = VAL(i, T) DO S END;
  IF i > LAST(INTEGER) - delta THEN EXIT END;
  INC(i, delta);
  IF i > done THEN EXIT END;
END
END

```

The two IF statements are almost adjacent. Only the *INC* statement is between them. In the next step the second IF statement and the *INC* statement will be reordered. As *i* appears in both statements, we need to be careful when reordering. To make it explicit which value of *i* is used at which time, let's rename *i* to *i_b* when it is before the *INC* statement and *i_a* when it is after the *INC*.

```

IF i_b > LAST(INTEGER) - delta THEN EXIT END;
i_a := i_b + delta;
IF i_a > done THEN EXIT END;
i_b := i_a;

```

Now we can see that the second condition can be transformed from $i_a > done$ to $i_b + delta > done$. This check can be moved before the *INC* statement, as the condition has become independent of *i_a*. Then we have:

```

IF i > LAST(INTEGER) - delta THEN EXIT END;
IF i + delta > done THEN EXIT END;
INC(i, delta);

```

The two conditions can be made similar by subtracting *delta* from both sides of the second condition.

```

IF i > LAST(INTEGER) - delta THEN EXIT END;
IF i > done - delta THEN EXIT END;
INC(i, delta);

```

As *done* can never be greater than *LAST(INTEGER)*, the first check can be omitted completely. Now we have one problem left, which is the possible overflow of the subtraction, which can happen when *done* is near *FIRST(INTEGER)*. In this case, *done* lies in the interval $[FIRST(INTEGER); FIRST(INTEGER) + (step - 1)]$ and the *statements* have been executed for that value. In the next iteration *i* would be greater than *done*.

It would be great if the overflow check could be done as a special case of the only condition in the loop, as this would save some instructions. If the right hand side of the condition could take the value $FIRST(INTEGER) - 1$, this would work. But it cannot.

For integer arithmetics, $a > b \equiv a \geq b + 1$ holds. So $i > done - delta$ can be rewritten as $i \geq done - (delta - 1)$. Now the special case can be integrated by making sure that the right hand side is equal to *FIRST(INTEGER)*. Let's call the right hand side *limit* and see which value it takes:

```

IF done < FIRST(INTEGER) + (delta - 1) (* integer overflow *)
  THEN limit := FIRST(INTEGER);
  ELSE limit := done - (delta - 1);
END;

```

This code only works well if $\delta \geq 1$. In the (unlikely) case that $\delta = 0$, *limit* should be set to *done*, so the loop terminates as soon as the *last* expression from the FOR statement is reached, not when it is *passed* (as specified in [M3def]).

Putting all things together, including the case of $\delta < 0$, the revised FOR statement looks like this:

```

FOR id := firstExpr TO lastExpr BY stepExpr DO statements END

```

is translated to

```

WITH first = firstExpr, last = lastExpr, step = stepExpr DO
  VAR
    var, limit: INTEGER;
  BEGIN
    var := first;
    IF ((step >= 0) AND (var <= last)) OR
      ((step < 0) AND (var >= last)) THEN

      IF step > 0 THEN
        IF last >= FIRST(INTEGER) + (step - 1)
          THEN limit := last - (step - 1);
          ELSE limit := FIRST(INTEGER);
        END;
      ELSIF step < 0 THEN
        IF last <= LAST(INTEGER) + (step + 1)
          THEN limit := last - (step + 1);
          ELSE limit := LAST(INTEGER);
        END;
      ELSE
        limit := last;
      END;

      LOOP
        WITH id = VAL(var, T) DO statements; END;
        IF ((step >= 0) AND (var >= limit)) OR
          ((step < 0) AND (var <= limit)) THEN EXIT; END;
        INC(var, step);
      END;
    END;
  END;
END;

```

5 Differences to the original FOR loop

- The new FOR loop is usable over the whole range of valid integers. No hidden overflow can occur.
- If $step = 0$ and $first = last$, the revised FOR loop exits after executing the *statements* once. The old FOR loop resulted in an endless loop.

6 Implementation costs

Inside the FOR loop, no additional code is needed. Before the loop starts, the value of *limit* must be calculated once.

- If *step* is known to be in the range $[-1; 1]$, the code needed to calculate *limit* reduces to nothing. Most uses of the FOR statement have $step = 1$, as this is the default value.
- If *last* and *step* are constant, the code reduces to nothing.
- If *step* is known to be non-negative, the negative branches can be optimized away.
- If *step* is known to be negative, the positive branches can be optimized away.

7 Conclusion

With a small runtime code addition, the FOR loops in Modula-3 can be made usable over the complete range of valid integers. As Modula-3 wants to be a nice, simple language, it should be considered if the overflow check is embedded into the language definition.

References

- [M3def] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson: *Modula-3: Language Definition*.
<http://research.compaq.com/SRC/m3defn/html/complete.html>