GNU Modula-2 update, catching semantic errors post code optimisation and improved debugging

Gaius Mulley
<gaius.mulley@southwales.ac.uk>

Department of Computer Science
University of South Wales
CF37 1DL
Current Status of GNU Modula-2

- current stable release is gm2-1.8.2 which grafts onto gcc-8.2.0
- it also builds on gcc-6.4.0, gcc-5.2.0, gcc-4.7.4 and gcc-4.1.2
- gm2-1.8.2 (on gcc-8.2.0), works well with automake, libtool and friends
Current Status of GNU Modula-2

- finished implementing soft integer overflow detection for addition, subtraction, negation and multiplication

- NaN soft runtime detection implemented
  - third party libraries can be easily added

- much of the code base has been reformatted to comply with GNU coding standards
“7. Release early. Release often. And listen to your customers.”
   - Eric S. Raymond, “The Cathedral and the Bazaar”

- New users complain about the use of uppercase keywords in Modula-2.

- New programmers complain that the debugger skips over lines of code.

- Mature programmers complain that the compiler should be able to detect more runtime errors at compile time, by utilising its optimisation knowledge.
  - Very tempting to ignore the first two complaints!
Emacs and a new Modula-2 mode

- inside gm2-1.8.2 there is a new Modula-2 emacs mode which needs to be pushed up to the emacs maintainers
  - users can configure keywords to be rendered lowercase/bold/underlined (any permutation allowed)
  - emacs saves the code with restored uppercase keywords

- credit to Benjamin Kowarsch for this idea
Debugger skipping over lines of code

- a common classroom complaint
- all compiler implementers know why this occurs, but new programmers seem to find this unnerving
- new option \texttt{-fm2-g} specifies at least one instruction will be issued for every statement keyword
  - \texttt{gm2} inserts a \texttt{nop} if there is no \texttt{tree} at the current location
Currently gm2 creates a tree representing `asm volatile ("nop")`

It would be really useful if the GCC middle/backend provided access to a platform independent `nop` tree.

Obviously it would trivial to add this for other front end languages.

Oddly two example programs actually went 3% faster with `-fm2-g` than without it on an AMD Black AMD FX(tm)-8350 Eight-Core Processor 4GHz.
Example videos of -fm2-g

- compiling with -g
  <http://floppsie.comp.glam.ac.uk/download/avi/gm2-gdb-normal-debugging.mp4>

- compiling with -g and -fm2-g
  <http://floppsie.comp.glam.ac.uk/download/avi/gm2-gdb-extended-debugging.mp4>
the ISO Modula-2 standard specifies that the following runtime checking is performed:

- indexException, rangeException, caseSelectException, invalidLocation, functionException, wholeValueException, wholeDivException, realValueException, realDivException, complexValueException, complexDivException, protException, sysException, coException, exException

- a number of these are detected within the ISO libraries and some are detected by the compiler
GNU Modula-2 runtime switches

- **-fnil**  NIL pointer violation.
- **-fwholediv** division by 0.
- **-findex** array indice out of range.
- **-frange** assignment range error ordinal types.
- **-freturn** procedure functions finishing without a return statement.
- **-fcase** missing case clause.
- **-fwholevalue** ordinal overflow detection on addition, subtraction, multiply and negate.
- **-ffloatvalue** NaN detection after every floating point operator.

- **-fwholevalue** and **-ffloatvalue** are new to gm2-1.8.2 (gcc-8.2.0)
Modula-2 is a strongly typed language and its arithmetic is type safe
- assignment relaxes type strictness (between signed and unsigned data types)
- basic operators (+, −, *, /, MOD and DIV) insist operands are of the same type

thus it is possible to detect when ordinal values are about to go out of range for +, −, * and unary – for any value for arbitrary user defined subranges

and it is is possible to detect specific value errors for /, MOD and DIV
GNU Modula-2 can detect and report which operator has caused an overflow.
in Modula-2 this is perhaps more useful than in C as users can declare subrange ordinal types

for example:

```plaintext
PROCEDURE foo ;
VAR
   x: [-1..4] ;
   y: INTEGER ;
BEGIN
   x := 2 ;
   y := -x  (* runtime error at this line. *)
END foo ;
```
General tests for ordinal overflow on unary minus
(pseudo code)

(* general purpose subrange type, i, is currently legal, min is MIN(ordtype) and max is MAX(ordtype). *)

PROCEDURE sneg (i: ordtype) ;
BEGIN
    max := MAX (ordtype) ;
    min := MIN (ordtype) ;
    IF (i#0) AND (* cannot overflow if i is 0 *)
       (* will overflow if entire range is positive. *)
       (((min >= 0) AND (max >= 0)) OR
        (* will overflow if entire range is negative. *)
        ((min <= 0) AND (max <= 0)) OR
        ((min < 0) AND (max > 0) AND ((min + max) > 0) AND (i > -min)) OR
        ((min < 0) AND (max > 0) AND ((min + max) < 0) AND (i < -max))
     )THEN
        error ("type overflow")
    END
END sneg ;
General tests for ordinal overflow on unary minus (pseudo code)

- this is then hand translated into tree code which builds a large expression tree containing the failure condition

  see checkWholeNegateOverflow in [http://git.savannah.gnu.org/cgit/gm2.git/tree/gcc-versionno/gcc/gm2/gm2-gcc/m2expr.c?h=gcc_8_2_0_gm2](http://git.savannah.gnu.org/cgit/gm2.git/tree/gcc-versionno/gcc/gm2/gm2-gcc/m2expr.c?h=gcc_8_2_0_gm2)
General tests for ordinal overflow on unary minus

```
gcc-versionno/gcc/gm2/gm2-gcc/m2expr.c:checkWholeNegateOverflow

...  
  tree o3 = Build4TruthAndIf (location, c7, c8, c9, c10);
  tree o4 = Build4TruthAndIf (location, c7, c8, c11, c12);

  tree a2 = Build4TruthOrIf (location, o1, o2, o3, o4);
  tree condition = FoldAndStrip (BuildTruthAndIf (location, a1, a2));

  tree t = BuildIfCallWholeHandlerLoc
    (location, condition, "whole value unary -");
  AddStatement (location, t);
  }
```
Revisiting the example foo

MODULE bar ;
PROCEDURE foo ;
VAR
  x: [-1..4] ;
  y: INTEGER ;
BEGIN
  x := 2 ;
  y := -x  (* runtime error at this line. *)
END foo ;
BEGIN
  foo
END bar.

$ gm2 -fsoft-check-all -g -fm2-g bar.mod
$ ./a.out
bar.mod:9:3:the whole value is about to overflow in whole value unary -
General tests for ordinal overflow on addition

- check to see whether \( i + j \) will overflow an ordinal type \((\text{ordtype})\)

- pseudo code

```pseudo
PROCEDURE sadd (i, j: ordtype) 
BEGIN
    IF ((j>0) AND (i > MAX(ordtype)-j)) OR 
        ((j<0) AND (i < MIN(ordtype)-j))
    THEN
        error ("signed addition overflow")
    END
END sadd 
```
General tests for ordinal overflow on addition

gcc-versionno/gcc/gm2/gm2-gcc/m2expr.c:checkWholeAddOverflow

```c
static void checkWholeAddOverflow (location_t location, tree i, tree j,
    tree lowest, tree min, tree max)
{
    tree c1 = BuildGreaterThanZero (location, j, lowest, min, max);
    tree c2 = BuildGreaterThan (location, i, BuildSub (location, max, j));
    tree c3 = BuildLessThanZero (location, j, lowest, min, max);
    tree c4 = BuildLessThan (location, i, BuildSub (location, min, j));
    tree c5 = FoldAndStrip (BuildTruthAndIf (location, c1, c2));
    tree c6 = FoldAndStrip (BuildTruthAndIf (location, c3, c4));
    tree condition = FoldAndStrip (BuildTruthOrIf (location, c5, c6));
    tree t = BuildIfCallWholeHandlerLoc (location, condition, "whole value +");
    AddStatement (location, t);
}
```
Check to see whether i - j will overflow an ordtype

pseudo code:

PROCEDURE ssub (i, j: ordtype) ;
BEGIN
    IF ((j>0) AND (i < MIN(ordtype)+j)) OR ((j<0) AND (i > MAX(ordtype)+j))
    THEN
        error ("signed subtraction overflow")
    END
END ssub ;
Check to see whether $i - j$ will overflow an ordtype

```c
static void
checkWholeSubOverflow (location_t location, tree i, tree j,
                     tree lowest, tree min, tree max)
{
  tree c1 = BuildGreaterThanZero (location, j, lowest, min, max);
  tree c2 = BuildLessThan (location, i, BuildAdd (location, min, j));
  tree c3 = BuildLessThanZero (location, j, lowest, min, max);
  tree c4 = BuildLessThan (location, i, BuildAdd (location, max, j));
  tree c5 = FoldAndStrip (BuildTruthAndIf (location, c1, c2));
  tree c6 = FoldAndStrip (BuildTruthAndIf (location, c3, c4));
  tree condition = FoldAndStrip (BuildTruthOrIf (location, c5, c6));
  tree t = BuildIfCallWholeHandlerLoc (location, condition, "whole value -");
  m2type_AddStatement (location, t);
}
```
Check to see whether $i \times j$ will overflow an ordinal type

PROCEDURE smult (i, j: ordtype) ;
BEGIN
  IF ((i > 0) AND (j > 0) AND (i > max DIV j)) OR
      ((i > 0) AND (j < 0) AND (j < min DIV i)) OR
      ((i < 0) AND (j > 0) AND (i < min DIV j)) OR
      ((i < 0) AND (j < 0) AND (i < min DIV j))
THEN
  error ("signed multiplication overflow")
END
END smult ;

if ((c1 && c3 && c4) ||
(c1 && c5 && c6) ||
(c2 && c3 && c7) ||
(c2 && c5 && c7))
error ("signed multiplication overflow")
Check to see whether \( i \times j \) will overflow an ordinal type

```c
static
void
checkWholeMultOverflow (location_t location, tree i, tree j,
    tree lowest, tree min, tree max)
{
    tree c1 = BuildGreaterThanZero (location, i, lowest, min, max);
    tree c2 = BuildLessThanZero (location, i, lowest, min, max);
    tree c3 = BuildGreaterThanZero (location, j, lowest, min, max);
    tree c4 = BuildGreaterThan (location, i, BuildDivTrunc (location, max, j));
    tree c5 = BuildLessThanZero (location, j, lowest, min, max);
    tree c6 = BuildLessThan (location, j, BuildDivTrunc (location, min, i));
    tree c7 = BuildLessThan (location, i, BuildDivTrunc (location, min, j));
    tree c8 = Build3TruthAndIf (location, c1, c3, c4);
    tree c9 = Build3TruthAndIf (location, c1, c5, c6);
    tree c10 = Build3TruthAndIf (location, c2, c3, c7);
    tree c11 = Build3TruthAndIf (location, c2, c5, c7);

    tree condition = Build4LogicalOr (location, c8, c9, c10, c11);
    tree t = BuildIfCallWholeHandlerLoc (location, condition, "whole value *");
    AddStatement (location, t);
}
```
Detecting integer overflow at runtime

```plaintext
MODULE multint1;

FROM libc IMPORT exit;

VAR
  i, j, k: [-8..7];
BEGIN
  i := 3;
  j := 3;
  k := i * j;
  exit (0) (* should not get here if -fsoft-check-all is used *)
END multint1.
```
Compiling with -fsoft-check-all

```
$ gm2 -fsoft-check-all -g multint1.mod
$ ./a.out
multint1.mod:28:3:the whole value is about to overflow in whole
value *
Aborted
```
Going further with exception

- A plugin has been written which detects 21 further categories within the ISO exception taxonomy.

- The plugin resides at the end of gimple optimisations:
  - `*warn_function_noreturn`
/* runtime exception inevitable detection. This plugin is most effective if it is run after all optimizations. This is plugged in at the end of gimple range of optimizations. */
pass_info.pass = make_pass_warn_exception_inevitable (g);
pass_info.reference_pass_name = "*warn_function_noreturn";

pass_info.ref_pass_instance_number = 1;
pass_info.pos_op = PASS_POS_INSERT_AFTER;

register_callback (plugin_name,
    PLUGIN_PASS_MANAGER_SETUP,
    NULL,
    &pass_info);
```
unsigned int
pass_warn_exception_inevitable::execute (function *fun)
{
    gimple_stmt_iterator gsi;
    basic_block bb;

    FOR_EACH_BB_FN (bb, fun)
    {
        for (gsi = gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
            runtime_exception_inevitable (gsi_stmt (gsi));
        /* we only care about the first basic block in each function. */
        return 0;
    }
    return 0;
}
```
Combining integer overflow and the exception handler plugin

```
MODULE multint1;

FROM libc IMPORT exit;

VAR
  i, j, k: [-8..7];
BEGIN
  i := 3;
  j := 3;
  k := i * j;
  exit (0) (* should not get here if -fsoft-check-all is used *)
END multint1.
```
Compiling with -fsoft-check-all -O

$ gm2 -fsoft-check-all -g -O multint1.mod
multint1.mod:28:3:inevitable that this error will occur at runtime, expression will generate an exception as a whole value will overflow the type range
Future work and conclusions

- clearly scope for improved messages
- technique works well!
- need to complete `DIV` and `MOD` integer overflow for arbitrary subranges
- most importantly, it is time to move gm2 into the gcc tree
- would be good to implement some of the m2r10 language proposals
  - extensible records and removing variant record for `-fm2r10`
- thank you to all GCC developers and the gm2 mailing list community