Compile-time unit checking in Fortran with genunits

Ben Trettel

FortranCon 2025
2025-11-05

Why check units in Fortran code?

Bugs

- Bugs in operations: The operation 1 kg + 1 m is meaningless and indicates there is a bug.
- Bugs in procedure arguments: You shouldn't pass in a kg when a m is expected.
- Unit checking identifies the location of bugs.
- Oracles sometimes don't exist or are a pain.
- Redundancy: If a test has bugs, unit checking might find bugs the test misses.

Documentation

- Checking units is enforced documentation, similar to assertions.

Obtaining genunits

- genunits is experimental, incomplete, and you probably don't want to use it.
- genunits can be obtained here: <https://github.com/btrettel/flt/>
  - `make FC=compiler_command genunits`

Motivation of talk

- Unit checking is typically applied to toy problems of relatively small scale.
- I set out to apply compile-time unit checking (to be discussed) to “production” code for a personal project and ran into major issues.
- This talk summarizes the issues found.
- In development code to test genunits: https://github.com/btrettel/blastersim
- Note: The compile-time approach here is not new and was probably first done by Brad Richardson: https://gitlab.com/everythingfunctional/quaff

genunits in action (1/4)

```
program test_units_pass

use units, only: si_length   => unit_p10_p00_p00_p00, &
                 si_time     => unit_p00_p00_p10_p00, &
                 si_velocity => unit_p10_p00_m10_p00
implicit none

type(si_length)   :: x
type(si_time)     :: t
type(si_velocity) :: v

x%v = 1.0
t%v = 1.0

v = x / t

print *, v

end program test_units_pass
```

Code is here: http://www.trettel.us/dl/genunits.zip

genunits in action (2/4)

Result (passing)

```
$ gfortran units.o test_units_pass.f90 -o test_units_pass
$ ./test_units_pass
 1.00000000 m/s
```

genunits in action (3/4)

```
program test_units_fail

use units, only: si_length    => unit_p10_p00_p00_p00, &
                 si_time      => unit_p00_p00_p10_p00
implicit none

type(si_length) :: x, v
type(si_time)   :: t

x%v = 1.0
t%v = 1.0

v = x / t

print *, v

end program test_units_fail
```

genunits in action (4/4)

Result (failing)

```
$ gfortran units.o test_units_fail.f90 -o test_units_fail
test_units_fail.f90:13:4:

   13 | v = x / t
      |    1
Error: Cannot convert TYPE(unit_p10_p00_m10_p00) to TYPE(
    unit_p10_p00_p00_p00) at (1)
```

genunits process

- The units module is generated by genunits and is ideally custom for each program.
- The process:
  1. An input file defining the **base units** (example: m, kg, s) and other configuration is read.
  2. genunits generates a set of units fitting the provided specifications and outputs a Fortran module.
  3. The generated module can then be used by Fortran code.

partial genunits input file

```
&config
output_file      = "units50.f90"
base_units       = "m", "kg", "s", "K"
type_definition  = "real"
use_line         = ""
kind_parameter   = ""
module_name      = "units"
max_n_units      = 50

!                  m      kg     s      K
min_exponents =  -3.0,  -1.0,  -2.0,  -1.0
max_exponents =   3.0,   1.0,   2.0,   1.0
denominators  =     1,     1,     1,     1

debug = .false.

dtio       = .true.
sqrt       = .true.
cbrt       = .false.
square     = .true.
intrinsics = .true.
/

&seed_unit
label = "unitless"
!   m    kg    s    K
e = 0.0, 0.0, 0.0, 0.0
/
```

The naive generator approach

- Naively, units can be generated by creating all combinations of base units (example: m, kg, s) within certain exponent ranges.
- For example, consider m with an exponent range of -1 to 1 and s with an exponent range of -1 to 1: $m^{-1} \cdot s^{-1}$, $m^{-1} \cdot s^{0}$, $m^{-1} \cdot s^{1}$, $m^{0} \cdot s^{-1}$, $m^{0} \cdot s^{0}$, $m^{0} \cdot s^{1}$, $m^{1} \cdot s^{-1}$, $m^{1} \cdot s^{0}$, $m^{1} \cdot s^{1}$
- This leads to a large number of unused units and slower compilation time than necessary.

Compilation time as a function of units

- gfortran 13.3.0 vs. nvfortran 25.9
- `$COMPILER -c -o units.o units.f90`
- Specific number of procedures not a unique function of number of units.
- Any code using the units module will also compile slowly.

units	procedures	gfortran time (s)	nvfortran time (s)
18	455	0.689	29.322
50	2359	3.923	3861.997
100	7983	24.527	—
200	31629	399.098	—

Units modules for this test can be downloaded here: http://www.trettel.us/dl/genunits.zip

genunits iteration process (1/2)

- Instead of naively generating all possible units in the desired range, genunits starts with **seed units** and generates a set of units that result from operations on those seed units.
- The seed units are typically the units used in variable declarations, but other units can be added if required.
  - I call units required but not explicitly used “intermediate units”.

genunits iteration process (2/2)

- The generation process is iterative:
  1. Create a set of units from the seed units.
  2. Next iteration creates new units from operations on the current set of units. Example: m and s with the division operator produces m/s.
  3. The process is continued until the desired number of units or iterations are obtained, or no more units are created.
- This process prioritizes the units created, often reducing compilation time. However, it is not a panacea as often the number of required units is still large. And unused units will still appear.

Performance impact: use inlining

- At first I thought that the performance impact was terrible.
- However, Walter Spector on the Fortran Discourse recommended enabling inlining.
- A Gauss-Seidel Poisson solver was implemented with both `reals` and genunits:
  - `ifx -O2`: genunits was 21.3 times as slow
  - `ifx -O2 -flto`: genunits was 1.37 times as slow
- This slows down compilation more but is worthwhile.

What would I like to see in the future?

- Ideas to reduce compilation time without improving compilers.
- Faster compilation of modules with a large number of types in compilers.
- Easier inlining in compilers.
- Units implemented in the Fortran standard, and/or a compiler.
  - Maybe a directive based approach like Camfort could be accepted by a compiler if units are not added to the standard?

Two approaches using derived types (1/2)

Run-time checking: one derived type

- Example declaration from PhysUnits: https://github.com/gpetty/PhysUnits
  - `type(preal) :: radius`
  - `radius = 6370.*u_kilometer`
- Unit checking is done at run-time.
- Testing with high code coverage is required to check units.
- Code lines with bugs can be identified from a backtrace or debugger.
- Performance is definitely decreased.

Two approaches using derived types (2/2)

Compile-time checking: many derived types

- Example declaration from quaff: https://gitlab.com/everythingfunctional/quaff
  - `type(length_t) :: distance`
- Unit checking is done at compile-time.
- Testing is not required to check units.
- Location of bugs can be identified by a compiler *within* a line.
- There is still a run-time performance decrease, but presumably it is less than with run-time checking.
- **This is the approach genunits takes.**

Intermediate units

Variable declarations:

- m: x, y, z
- m^3: V

$$V = x \cdot y \cdot z$$

But: $x \cdot y$ has units of m^2, which does not appear in the declarations. So defining types for only the units that appear in the variable declarations is insufficient. Intermediate units, for example, m^2, need to be defined.