An Intermediate Representation for Structured Input

October 17, 2016

L.E. Busby

LLNL-PRES-703906
This work was performed under the auspices of the U.S. Department of Energy, by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Lawrence Livermore National Laboratory

IREP, The General Idea

In operation, IREP:

— Reads program input in the form of Lua tables;

— Places the input values into compiled, structured variables, available from either C/C++ or Fortran, or both together;

— Detects and reports many simple input errors, automatically;

— Can read most any sort of *plain old data* without guidance, or anything else, with some extra effort.

To set it up:

— You define a *data store*, a set of *well-known tables*, the “compiled, structured variables” mentioned above;

— This is essentially the same as writing a set of nested C structs, or Fortran derived types;

— The difference is that the “structs” are written using simple cpp(1) macros, one line in, one line out;

— That’s it: No other wrapping nor metaprogramming is needed.

Can IREP be Useful to Your Code?

1. IREP is a good way to handle initial problem setup for most common input data;
2. It contains scalar and 1-D integer, double, logical, and string variables;
3. It also gives you Lua callback functions, if you want them;
4. Defining one variable pretty much takes one line of code;
5. It can work with C/C++ or Fortran codes, or both together, sharing a common data store;
6. Defining the IREP data store is pretty easy, best done by a domain specialist (not a computer scientist);
7. Reading an entire Lua table generally takes one line of code;
8. IREP is fairly small: About 350 lines of code in the basic system, plus your tables;
9. Other than that, it doesn't do much.

From Zero to IREP in Eight Steps

1. Lua table constructors are very nice:
   ```
   t = {
     t1 = {
       a = 3
     }
   }
   ```
2. Each table element has a dual representation: ***`t.t1.a=3`***;
3. It's easy to make a reader to convert from (1) to (2);
4. Form (2) looks just like a reference to a C struct;
5. The ISO_C_BINDING maps C to Fortran: ***`t.t1.a=3`*** ⇔ ***`t%t1%a=3`***
6. Fortran *read namelist* can parse strings like ***`"t%t1%a=3"`***;
7. The C preprocessor can output either C or Fortran from one input;
8. So we can read a Lua table, make a Fortran string, parse it with *read namelist*, and put the result in one spot available to both the C/C++ and Fortran code.

Lua Table ≈ C Struct ≡ Fortran Derived Type

Here is a table as it might appear in the three languages:

```
Lua                C/C++              Fortran
---                -----              -------
t = {              struct irt_t {     type, bind(c) :: irt_t
  a = 3,             int a;             integer(c_int) :: a=3
  b = 7.2,           double b;          real(c_double) :: b=7.2
  s = "abc",         char[8] s;         character(c_char) :: s(8)="abc"
}                  };                 end type
```

And here is the IREP definition for the same table:

```
Beg_struct(irt_t)
  ir_int(a,3)        // Integer named ‘‘a’’, default value 3.
  ir_dbl(b,7.2)      // Double named ‘‘b’’, default 7.2.
  ir_str(s,8,"abc") // String ‘‘s’’, maxlen 8, default "abc".
End_struct(irt_t)
```

The Lua Table Reader is Simple

```
Lua input          Fortran-compatible output
---------          -------------------------
t = {
  a = 3,       →   "t%a = 3"
  b = 7.2,     →   "t%b = 7.2"
  s = "abc",   →   "t%s = 'abc'"
}
```

The output of the table reader is a list of strings, each of which “looks like” a Fortran assignment. These strings are fed to the *assignment parser*, (a Fortran *namelist* statement), which loads it into the IREP data store.

If there is a matching variable in the data store, all is well. If not, it’s an error. (This will catch a large class of input errors, and is a significant benefit.)

So the IR data store effectively *defines* the set of Lua input tables that can be recognized by the reader. Such tables are called *well-known tables*.

The Assignment Parser (The Whole Thing)

```
/* The C part calls a Fortran function ir_rd_nml. */
int read_nml(const char *frep, const char *val) {
  char buf[BSZ];
  extern int ir_rd_nml(char *, int);
  (void)snprintf(buf, BSZ, "&ir_input %s = %s /", frep, val);
  return ir_rd_nml(buf, (int)strlen(buf));
}
```

```
! The Fortran function:
integer(c_int) function ir_rd_nml(s,n) bind(c)
  integer(c_int), value, intent(in) :: n
  character(len=1,kind=c_char), intent(in) :: s(n)
  character(kind=c_char,len=n) :: fs
  namelist /ir_input/ table1, table2, ...
  fs = transfer(s,fs)
  read(fs, nml=ir_input, iostat=ir_rd_nml)
end function
```

All the work is done by the *read(fs, ...)* statement. The *namelist* statement ***registers*** the variables *table1*, *table2*.

Limitations

1. IREP stores “plain old data”, strictly typed, along with Lua callback functions that accept double precision parameters and return double precision values.
2. There is an escape mechanism to store other sorts of things, but that does take some work.
3. C/Fortran interoperability rules limit IREP to static objects: Array sizes, including string lengths, are defined at compile time;
4. Strings, though standard, are neither idiomatic “C” nor “Fortran”. (An equal opportunity annoyance.)
5. Names in the IR follow the Fortran rule of case insensitivity.
6. The data store has no introspection, nor bulk operations.
7. From the Lua table reader’s perspective, the data store is “write-only”.
8. You need to use both C/C++ and Fortran compilers to build IREP.

IREP in the MAPP Code Project

- We have built an IREP data store containing 425 variables in 740 LOC, spread over 62 structures and 10 well-known tables.
- About 94% are scalars, the rest are vectors of various lengths.
- It also contains about 64 Lua callback functions, with 3−5 parameters, and 1−3 return values.
- Nearly all the project-specific code (the structure and contents of the well-known tables) has been written by the code physicists. It does not require a code specialist.
- The project centers around a collaboration between a C++ code and a Fortran code, that read a common input format. IREP has provided a simple, crisp way for the collaborators to *define* agreement on the input and keep moving forward.

Example: The *initial_conditions* well-known table

```
#include "ir_start.h"

Beg_struct(irt_ic_material)
  ir_int(region,-1)
 Callback_dd(density)
 Callback_dd(energy)
 Callback_dd(volume_fraction)
 Callback_dd(burn_fraction)
End_struct(irt_ic_material)

Beg_struct(irt_initial_conditions)
  ir_dbl(start_time,0.0e0)  // Time at which to start simulation
  ir_dbl(stop_time,1.0e99)  // Time at which to stop simulation
  ir_dbl(time_step,1.)      // Initial time step
 Vstructure(irt_ic_material,material,0:98,99)
 Callback_dd(magnetic_field)
 Callback_dd(velocity)
 Callback_dd(temperature)
 Callback_dd(ion_temperature)
End_struct(irt_initial_conditions)

// Declare it.
ir_wkt(irt_initial_conditions, initial_conditions)

#include "ir_end.h"
```

Example: Lua *initial_conditions* problem input

```
local rho0, e0, eps = 1, 0, 1.e-6

initial_conditions = {
  start_time = 0,
  stop_time = 0.6,
  velocity   = function(x,y,z)
    local r = math.sqrt(x^2 + y^2 + z^2)
    if (r > eps) then
      return -x/r, -y/r, -z/r
    else
      return 0,0,0
    end
  end,
  material = {
    [1] = {
      density = rho0, -- A constant function.
      energy = e0,
      volume_fraction = 1.0,
    }
  }
}
```

Availability

- Lee Busby: busby1@llnl.gov
- https://github.com/LLNL/irep

```
This is the README for IREP.  See also etc/doc/ for additional
documentation.

The Intermediate Representation (IR) is a tool for constructing a set
of C/C++ and Fortran data structures, and a tool for reading Lua tables
into those structures.  It is built around the observation that the
textual representation of Lua table elements can frequently be mapped
directly into a C/C++ struct, or a Fortran derived type.  Suppose a Lua
table constructor is given as follows:

  t1 = {
    t2 = {
      x = 42,
    }
  }

With appropriate prior definitions, we could alternatively write:

  t1.t2.x = 42 -- Lua
  t1.t2.x = 42 // C/C++
  t1%t2%x = 42  ! Fortran
  [ ... ]
```