ECL Dependency Project Notes

2022-02-03

The research goal of the ECL dependency project is to track data usage within an ECL program. More specifically, the result of a usage analysis should make it possible to forward-trace input values (scalar or otherwise) and see if/how they affect any output; and it should be possible to back-trace output values (scalar or otherwise) to see their source(s).

There are three general items to track:

* Basic field dependencies (inputFile.fieldA -> outputFile.fieldB).  The term 'field' should encompass all types of data read at runtime, not just whatever is in a file, and may include pure scalar values.
* Whether the previous is "dirty" vs. "clean".  A clean dependency would be a straight copy of the source's value.  If the value runs through any transformation or even a function, mark it as dirty.
* Track attributes that affect output fields.  A transform statement like 'SELF.foo := IF(someValue, LEFT.fieldA, LEFT.fieldB)' would cause SELF.foo to depend on the boolean value of someValue even though someValue’s value does not appear in SELF.foo. Attributes that affect output like this should themselves be tracked backwards to identify their dependencies.

The result of the analysis could be one or more directed acyclic graphs (DAGs). Nodes should correspond to ECL attributes, fields, or functions (including locations within source code, if applicable). Edges should correspond to data movement, and be marked with dirty/clean indicators.

2022-02-04: The eclcc client-side command-line tool is being modified to emit an intermediate representation (IR) of parsed ECL. This IR fully identifies attributes, functions, transforms, etc. A tool will need to be written to parse the IR information and convert it into the DAGs.

2022-02-08: Existing documentation on the ECL IR format can be found at the top of the C++ source code file ecl/hql/hqlir.cpp.

ECL programs of ever-increasing complexity will be devised and submitted to the toolset for analysis. Once a given complexity is conquered, new ECL programs will be created. In this way, edge cases and complex data movements may be tackled in a step-wise manner.

A simple (though not the simplest) ECL program follows for illustrative purposes.

ECL Example

IMPORT Std;

SourceDataRec := RECORD

STRING person\_name;

STRING birthday\_yyyymmdd;

END;

sourceData := DATASET

(

'~test::sample\_person\_data',

SourceDataRec,

FLAT,

OPT

);

OutDataRec := RECORD

SourceDataRec;

UNSIGNED2 age;

END;

today := Std.Date.Today() : INDEPENDENT;

outData := PROJECT

(

sourceData,

TRANSFORM

(

OutDataRec,

bday := Std.Date.FromStringToDate(LEFT.birthday\_yyyymmdd, '%Y%m%d');

SELF.age := Std.Date.YearsBetween(bday, today),

SELF := LEFT

)

);

OUTPUT

(

outData,

{outData},

'~test::sample\_person\_data\_expanded',

OVERWRITE, COMPRESSED

);

Data Flow Diagram



Blue lines indicate “clean” copies of data. Red lines indicate “dirty” copies of data. Not shown here, but green lines would indicate attributes that affect downstream data but don’t become downstream data themselves (see the SELF.foo example on the first page).

The eclcc client-side binary (available within the HPCC Client Tools library) can emit attribute information for a well-formed ECL program. The following command produces XML output on stdout, so it can easily be captured by calling applications:

eclcc -M MyFile.ecl

An example output – which can become quite large – can be provided. The result using the example ECL code above runs over 3000 lines of output.

It is unknown if using eclcc in this manner will be useful or not for this project. I am providing it merely has an additional piece of information.