Translating LPJ-Guess from C++ to Futhark

Ulrik Elmelund

Advisor: Cosmin Oancea

Submitted: 2022-06-24

Contents

1 Background

1.1 LPJ-GUESS

LPJ-GUESS is a framework for modelling terrestrial ecosystems at various scales.[1] It models various different ecological systems, such as photosynthesis, respiration and stomatal regulation at high temporal resolution, with slower processes such as growth, population dynamics and disturbance at larger time steps. Input parameters involve climate parameters, atmospheric carbondioxide concentrations and a soil parameter. Daily air temperature, precipitation and a parameter for sunshine are also involved. All of these things, and more, come together to model the dynamics of plant growth in small or large areas over spans of years.

The models constructed with the framework require intensive data-parallel computations, which lends itself to efficient GPU execution.

Translating the existing industry-size and quality framework, written in idiomatic sequential imperative C++ code, into a language which makes GPU execution both possible and efficient would be desireable for the scientific community, as the performance-gains might be multiple orders of magnitude.

Futhark is a morally functionally pure data-parallel array language, currently under development at diku, which generates efficient GPU code[2].

1.2 Project

The project consist in investigating the feasability of translating the framework, written in sequential imperative code into Futhark.

This will be quite challenging, as the existing implementation is very complex, and as I do not have much experience with large professional codebases of this kind.

If time permits, there will also be performance evaluation of the resulting futhark implementation. However, the main burden of the project is software engineering of translating the code.

Practically, it involves getting familiarised with the existing codebase, its structure, and creating a mental model of this. Based on this understanding, a plan for which parts to initially emulate with futhark can be developed, and then as a start to translating the whole framework, a futhark equivalent of the main moving parts of the framwork can be constructed. Initially this can be tested using extracted input and output values from the framework as a reference, and eventually integrated directly. Futhark is only relevant for the parts of the framework that do the heavy lifting, and can be integrated to replace those parts in the existing framework.

1.3 Main learning goals

1. Getting familiarised the software engineering and architecture of a large industry simulation framework.
2. Investigate the challenges and feasability of translating idiomatic imperative C++ code to the functional array language Futhark.
3. Investigate how to prove, test or argue for functional equivalence between a program and its manual translation.

1.4 Preliminary analysis and scope limitation

It is very relevant to read the architecture section of the technical document, and to read the paper[1] by Ben Smith to get familiarized with the overall structure of the existing implementation.

In order to focus on translating the functions in which most of the computational work is done first, and the dependencies of these. A good place to start is the `gprof` analysis:

```
Each sample counts as 0.01 seconds.
%         total
time      s/call  name
26.85     assimilation_wstress()
12.17     photosynthesis()
9.31      cnstep()
3.57      npp()
3.27      fpar()
2.68      leaf_phenology()
2.55      Soil::hydrology_lpjf()
2.54      Stand::is_highlatitude_peatland_stand()
2.34      decayrates_century()
2.34      ndemand()
```

I began by translating `assimilation_wstress()`, `photosynthesis()` and those functions and data structures it depended on.

As the project progressed, the `canopy_exchange()` function became the target of the present translation work.

2 Translation work

In order for the resulting conventions and code of the present project to be made the most useful for a follow up project, certain considerations are relevant to take into account.

There is a significant difference between writing Futhark code from the bottom up, and to writing it as a translation from an object oriented language - C++ in this case. This author does have some experience in writing Futhark programs, and likewise in writing C and C# programs, but not much experience in translating code in general, even less so in translating idiomatic code between paradigms.

It seems plausible that a hypothetical continuator of the present work is in a similar situation, and for that reason, I will present the reasoning behind the present work for consideration.

2.1 Data organisation

In translating the architecture, several choices presented themselves. First off, preserving the folder and file structure in translating the framework seems to have no drawbacks, and is very useful for keeping track of progress, comparing the translated code with its referent, etc. Given the dependency segregation principle of the SOLID principles of OOP, C++ has a distinction between header files (.h) and implementation files (.cpp) which is absent in futhark. Header files contain the definitions or interfaces of functions and classes, and the implementation files contain their implementations. This is not a strict distinction - the compilers will not force you to actually structure the code in this way - but it will pair headers and their corresponding implementation files together when compiling and linking the code.

Because of this one-to-one correspondece headers have with their bodies, I decided to conceptually combine them into single futhark files. So, for canexch.h and canexch.cpp, only canexch.fut will exist. For the purpose of keeping track of progress and such, the content from the header file should appear first, and their sepparation should be denoted by a visually substantial comment.

Whenever there is a name for something, a file, field or function, or anything else, it is desirable to preserve this name in the translation. When functions are overloaded, or have default values, and in many other situations, a reasonable alternative must be found.

2.2 Branches

```
1 if (!negligible(plai_grass))
2   flai=indiv.lai_today()/plai_grass;
3 else
4   flai=1.0;
```

Becomes

```
let flai =
  if (!negligible(plai_grass)) then
    lai_today(indiv, pfts[indiv.pft_id], ppfts[indiv.pft_id])/plai_grass
  else 1.0
```

Instead of a conditional branching to two assignments, we have one assignment to the outcome of a conditional, which just branches to two values. This is both easy to read and to write.

2.3 Basic values and operations

Most of the basic arithmetical operations used in the framework are common to both languages, as are basic data types. Doubles from C++ are represented by reals, which are parameterised and may be f64s or f32s for instance. Similar goes for ints. C++ structs can be represented most simply using records, and have almost equivalent semantics.

In many places of the code, small arrays of constant length are present. This is considered harmful[3], and is a possible target for future optimization.

2.4 Enumerators

In C++, enumerators are types with a very limited number of possible values, which are named tokens whose underlying representation is of secondary importance. However, they are essentially represented as integer constants, with each value of the enumerator, in order, having a value starting at zero and going up. For this reason, the last value of the enumerator is sometimes a pseudo-value, called for instance, NUM_SOME_ENUMERATOR. This allows the programmer to reference the size of the enumerator space, so to speak, but to me it seems like a hack. Printing an enumerator value to standard out will yield its underlying integer representation.

Futhark has built-in enumerator types which can contain arbitrary data. This is very flexible, however they can not be type-unified with integers. I went back and forth between the elegant native futhark representation, and the unelegant integer constant representation, but due to the seeming infeasability of printing the code representation of an enumerator in C++, i have currently gone for the integer constant representation. While this unfortunately is slightly more verbose in the futhark source code, it makes testing much more streamlined and productive, as one does not have to write unique print functions for each enumerator.

2.5 Functions, calling conventions and memory management

A widespread and useful convention in C++ is to have void functions that do not return anything, and that instead have return values of functions be allocated by the caller and passed to the callee by reference. C++ objects are likewise always passed by reference implicitly, whereas structs are passed by value unless otherwise specified.

Futhark passes everyting as values, and has no concept of reference or manual memory management, such as the C++ destinction between declaration and initialization.

Write-first values should be privatized, and only those values which have been modified should be returned. These values should then be shadowed in the caller, syntactically simulating them having been overwritten (which is morally impossible in a pure functional language).

2.6 Classes and their objects

Classes and objects do not exist in Futhark, so there is no one-to-one translation to be had. A reasonable convention therefore has to be discovered. In general, it is not desired for the translation to be a simulation of the original, but a functionally equivalent implementation of the original in the target language, true to the target language. One must therefore distinguish between objects and classes that exist to satisfy the OO paradigm, and objects and classes that exist to represent something in the mathematical model, of which the framework is an implementation.

To keep things simple, classes are represented as records, member methods as functions on records, and constructors as functions returning records.

Member functions should refer to the 'object' they are operating on as 'this'.

Class definitions:

```
1 class SomeClass {
2 public:
3   bool somefield;
4   double someotherfield;
5   // all the other class members here
6 }
```

```
type SomeClass = {
  somefield: bool,
  someotherfield: real
}
```

Constructors:

```
public
  SomeClass() {
    somefield = true;
    someotherfield = 2.0;
  }
```

```
let SomeClass() : SomeClass =
  {somefield = true
  ,someotherfield = 2.0
  }
```

Method:

```
double someMethod() {
  if(somefield){
    return someotherfield;
  } else {
    return someotherfield/1.0;
  }
}
```

```
let someMethod(this: SomeClass) : SomeClass =
  if(this.somefield)
    then this.someotherfield
    else this.someotherfield/1.0
```

Another method:

```
void anotherMethod(double somevalue) {
  someotherfield *= somevalue;
}
```

```
let anotherMethod(this: SomeClass, somevalue: real) : SomeClass =
  this with someotherfield = someotherfield * somevalue
```

Notice the verbosity of the in-place update of records. This is mostly an aesthetic concern, but when we get to functions that modify many fields of records many times, this becomes an issue, which brings us to:

2.7 Handling many in-place updates

OOP generally involves heavy modification of fields of objects, and a translation to futhark easily ends up having an equal number of in-place updates to records, which ends up being difficult to read.

It is advisable to extract values that are to be read multiple times into local variables, and it is advisable to generate and modify fields as local variables, externally, and then doing a series of in-place updates or a literal record construction at the end of the function. This keeps the code cleaner and more easily readable. Example, compare the following snippets for readability:

```
-- 1) Inundation stress
let ps_result = ps_result with agd_g = (ps_result.agd_g * inund_stress)
let ps_result = ps_result with rd_g = (ps_result.rd_g * inund_stress)

-- 2a) Moss dessication
let ps_result = if (pft.lifeform == MOSS)
  then
    let ps_result = ps_result with agd_g = (ps_result.agd_g *
↪  moss_ps_limit)
    let ps_result = ps_result with rd_g = (ps_result.rd_g *
↪  moss_ps_limit)
    in ps_result
  else ps_result
```

Compare the readability of the above code to:

```
-- 1) Inundation stress
let agd_g = agd_g * inund_stress
let rd_g = rd_g * inund_stress

-- 2a) Moss dessication
let (agd_g, rd_g) = if (lifeform == MOSS)
  then (agd_g * moss_ps_limit, rd_g * moss_ps_limit)
  else (agd_g, rd_g)
```

The first example (point 1) should suffice to motivate just avoiding doing inplace updates as much as possible, but notice how it becomes especially unwieldy when doing multiple conditional updates.

```
-- returning after having done verbose inplace updates
  in ps_result

-- returning by constructing a literal of the record
  in {agd_g = agd_g, rd_g = rd_g}

-- returning by constructing a literal of the record
  let ps_result = ps_result with agd_g = agd_g
  in ps_result with rd_g = rd_g
```

No matter the size of the record, constructing a literal is the simplest way to do it. The only case where it is advisable to do inplace updates before returning is if there are a lot of fields with values that have not been modified, and which should be preserved. Only in that case would you end up with shorter and more readable code by doing inplace updates.

However, that is actually what is happening most of the time - most of the time, you are modifying a few fields in very large records. Futhermore,

keeping track of which fields have been extracted and must be put back is errorprone. Doing inplace updates will not harm the performance of the code, since records are just sugar that disappears.

2.8 Updating arrays in records

Consider the following record:

```
type Fluxes = {
  annual_fluxes_per_pft : [npft][NPERPFTFLUXTYPES]real,
  monthly_fluxes_patch : [12][NPERPATCHFLUXTYPES]real,
  monthly_fluxes_pft : [12][NPERPFTFLUXTYPES]real,
  daily_fluxes_patch : [365][NPERPATCHFLUXTYPES]real,
  daily_fluxes_pft : [365][NPERPFTFLUXTYPES]real
}
```

Lets say we want to update three of these inner arrays. Ideally, we would like a simple syntax like this:

```
let report_flux_PerPFTFluxType_verbose2
    ( fluxes: Fluxes
    , flux_type: PerPFTFluxType
    , pft_id: int
    , value: real
    , month: int
    , day: int) : Fluxes =
      let fluxes.annual_fluxes_per_pft[pft_id, flux_type] += value
      let fluxes.monthly_fluxes_pft[month, flux_type] += value
      let fluxes.daily_fluxes_pft[day, flux_type] += value
      in fluxes
```

Hower, it is not possible to do this. Ditching the addition-assignment operator, and using the current `with` syntax, we get the following error:

```
Types
  [dim ][dim ]src
and
  {annual_fluxes_per_pft: [npft][NPERPFTFLUXTYPES]f64,
   daily_fluxes_patch: [365][NPERPATCHFLUXTYPES]f64,
   daily_fluxes_pft: [365][NPERPFTFLUXTYPES]f64,
   monthly_fluxes_patch: [12][NPERPATCHFLUXTYPES]f64,
   monthly_fluxes_pft: [12][NPERPFTFLUXTYPES]f64}
do not match.
```

adding some parenthesis solve this error, and we get the following rendering to the same effect as the one intended above. Lets also just look at the function body, since the signature is the same:

```
let fluxes = fluxes with annual_fluxes_per_pft =
  (fluxes.annual_fluxes_per_pft with [pft_id, flux_type] =
    fluxes.annual_fluxes_per_pft[pft_id, flux_type] + value)
let fluxes = fluxes with monthly_fluxes_pft =
  (fluxes.monthly_fluxes_pft with [pft_id, flux_type] =
    fluxes.annual_fluxes_per_pft[pft_id, flux_type] + value)
let fluxes = fluxes with daily_fluxes_pft =
  (fluxes.daily_fluxes_pft with [day,flux_type] =
    fluxes.daily_fluxes_pft[day,flux_type] + value)
in fluxes
```

This is a lot less readable, however, it is still not permitted. We get the following error:

```
Error at example.fut:218:8-13:
Using variable "fluxes", but this was consumed at 219:8-220:54.
↪  (Possibly through aliasing.)
```

This is a very interesting error. The inner update shadows the definition of fluxes, which can then not be shadowed in the outer update, it seems. This might also explain the size-type error. The language does not seem to expect records of a depth greather than one, or to support these. Two solutions considered are to forget in-place updates and just construct a new record, in this manner:

```
{ annual_fluxes_per_pft =
    annual_fluxes_per_pft with [pft_id, flux_type] =
      annual_fluxes_per_pft[pft_id, flux_type] + value
, monthly_fluxes_patch
, monthly_fluxes_pft =
    monthly_fluxes_pft with [month, flux_type] =
      monthly_fluxes_pft[month, flux_type] + value
, daily_fluxes_patch
, daily_fluxes_pft =
    daily_fluxes_pft with [day,flux_type] =
      daily_fluxes_pft[day,flux_type] + value
}
```

This will work, but it results in very verbose code for larger records.

The solution chosen is to use `copy` whenever there is a consumption-related error, in the following manner:

```
let fluxes = fluxes with annual_fluxes_per_pft =
  (copy fluxes.annual_fluxes_per_pft with [pft_id, flux_type] =
    fluxes.annual_fluxes_per_pft[pft_id, flux_type] + value)
let fluxes = fluxes with monthly_fluxes_pft =
  (copy fluxes.monthly_fluxes_pft with [pft_id, flux_type] =
    fluxes.annual_fluxes_per_pft[pft_id, flux_type] + value)
in fluxes with daily_fluxes_pft =
  (copy fluxes.daily_fluxes_pft with [day,flux_type] =
    fluxes.daily_fluxes_pft[day,flux_type] + value)
```

This gets the code past the checker. The danger then is that this might lead to redundant copying. If we compile this version and compare it to the one where we construct a literal of an updated record, using `futhark dev -s test.fut`, we can see that the two versions compile to effectively identical code. This is probably because intitial optimization steps of the compiler will recognize the copying step as redundant, and optimize it away. Since we are shadowing a value using a modified copy of itself, the original value will never be referenced again, only the modified copy. The optimizer sees this, and just modifies the original data instead of the copy.

2.9 A common map reduce mattern

Consider the following common pattern. We modify each member of a collection of values, and we perform some kind of accumulation over the modified values:

```
double fpar_tree_total=0.0;

vegetation.firstobj();
while (vegetation.isobj) {
  Individual& indiv=vegetation.getobj();

  // For this individual ...

  if (indiv.pft.lifeform==GRASS || indiv.pft.lifeform==MOSS) {

    // ...

    indiv.fpar=max(0.0,fpar_grass*flai-max(fpar_ff*flai,fpar_min));

    // ..
  }

  if (indiv.pft.lifeform==TREE) fpar_tree_total+=indiv.fpar;

  vegetation.nextobj();
}
```

We can translate this to a map followed by a reduce. However, we have to bind the result of the map, to shadow the old values, so the changes persist. Simply piping the result into the reduction will not make the modifications persistent:

```
let (vegetation) = map (\indiv ->
  if (pfts[indiv.pft_id].lifeform==GRASS ||
↪  pfts[indiv.pft_id].lifeform==MOSS) then

    -- ...

    let indiv = indiv with fpar =
↪  max(0.0,fpar_grass*flai-max(fpar_ff*flai,fpar_min))

    -- ...

    in indiv
  else (indiv)
  ) vegetation
let fpar_tree_total : real =
  reduce (+) 0.0 <| map
    (\i -> if pfts[i.pft_id].lifeform==TREE then i.fpar else 0.0
    ) vegetation
```

2.10 Type checker helpers

Consider the `init_canexch` function:

```
let init_canexch(patch : Patch, climate : Climate, vegetation :
↪  [npft]Individual, date : Date, pfts: [npft]Pft) : (Patch, Vegetation)
↪  =
  let vegetation =
    if (date.day == 0) then
      map (\indiv ->
        let indiv = indiv with anpp = 0.0
        let indiv = indiv with leafndemand = 0.0
        let indiv = indiv with rootndemand = 0.0
        let indiv = indiv with sapndemand = 0.0
        let indiv = indiv with storendemand = 0.0
        let indiv = indiv with hondemand = 0.0
        let indiv = indiv with nday_leafon = 0
        let indiv = indiv with avmaxnlim = 1.0
        let indiv = indiv with cton_leaf_aavr = 0.0
        in
        if (!negligible(indiv.cmass_leaf) &&
↪  !negligible(indiv.nmass_leaf)) then
          indiv with cton_leaf_aopt = indiv.cmass_leaf / indiv.nmass_leaf
        else
          let indiv = indiv with cton_leaf_aopt =
↪  pfts[indiv.pft_id].cton_leaf_max
          let indiv = indiv with mlai = replicate 12 realzero
          let indiv = indiv with mlai_max = replicate 12 realzero
          in indiv
      ) vegetation
    else vegetation
  in (patch with wdemand_day = 0, vegetation)
```

This implementation fails with the following error: (referencing the first inner binding)

```
Error at futsource/modules/canexch.fut:1674:21-41:
Full type of
  indiv
is not known at this point. Add a type annotation to the original record
↪  to
disambiguate.
```

If we modify the line with the `map` in this manner, to add a type annotation, in this manner:

```
map (\indiv : Individual ->
```

We still get the same error. This same error pattern is present in many places in the code. The current solution is to write functions of this kind:

```
let type_checker_helper8 (i: Individual) : Individual = i
```

And applying it in this manner:

```
let vegetation =
  if (date.day == 0) then
    map (\indiv ->
      let indiv = type_checker_helper8(indiv)
      let indiv = indiv with anpp = 0.0
      let indiv = indiv with leafndemand = 0.0
```

It seems that it fails to derive the type of the mapped individual from the collection mapped over, at least in these kinds of cases. The outer array being mapped over is type annotated in the function signature, so there should be no real ambiguity.

2.11 The classes, structs and their logical relations

The original architecture featured both inheritance and composition, and circular dependencies. All of this is troublesome to render in futhark, if not impossible. A simple solution to this is to abolish the circular references, and replace all references to parent objects with indices into global arrays.

Many objects are contained in parent objects, and also hold references to the same parents. They often access fields of parents, and parents call methods on children, and or access their fields. This circularity is only possible with references, which Futhark does not have. Even worse, the c++ code often has constructors of objects call constructors for children, the same constructors refering to fields in their parents.

There is no simple solution for this. The current solution is for the parents to pass the necessary fields to their childrens constructors, rather than references to themselves.

In order to represent the same data structures in Futhark, they must be flattened. In order to flatten them, it is necessary first to describe their mutual relations. Some are one-to-one, meaning that for every A, there exists one and only one B, and these reference each other. Some are one-to-many, for example Gridcells contain some number of Stands, and these all contain a

reference to their parent Gridcell. Still others are one-to-many-kinds, such as Climate, which are shared between several objects of different classes.

One-to-one objects can be combined or composed as is done now. All the others should be offloaded to a global state, and all references should be indirect, as indices into a global array, rather than a pointer.

My prediction of the contents of a flattened global data structure:

- `[p]Pft`
- `[]StandType`
- `[]ManagementType`
- `[s]Soiltype`
- `[n]Gridcell`
- `[n][p]Gridcellpft`
- `[n][s]Gridcellst`
- `[n]Climate`
- `[n]WeatherGenState`
- `[n]Landcover`
- `[n]MassBalance`
- `[n][m]Stand`
- `[n][m][p]Standpft`
- `[n][m][a]Patch`
- `[n][m][a]Soil`
- `[n][m][a]Fluxes`
- `[n][m][a][p]Patchpft`
- `[n][m][a][v]Individual`

Where n is the number of Gridcells, m is the number of Stands per Gridcell, p is the absolute number of Pfts, s is the number of Soiltypes, a is the number of Patches per stand, which may involve padding and v is the number of Individuals per Patch.

2.12 The functions and their logical relations

```
framework calls
  canopy_exchange (inlineable)
  ... and others

canopy_exchange calls
  init_canexch (inlineable)
  fpar (inlineable)
  photosynthesis_nostress (inlineable)
  ndemand (inlineable)
  vmax_nitrogen_stress (inlineable)
```

```
    wdemand (inlineable)
    aet_water_stress (inlineable)
    water_scalar (inlineable)
    npp (inlineable)
    leaf_senescence (inlineable)
    forest_floor_conditions (inlineable)

  photosynthesis is called by
    photosynthesis_nostress thrice
    vmax_nitrogen_stress twice
    wdemand
    assimilation_wstress twice

  assimilation_wstress is called by
    npp
    forest_floor_conditions
```

These are the most substantial functions in the calltree above `canopy_exchange` (ie. a large branch grows smaller twigs, which grow upwards from it). A lot of inlining can be done. The function `aet_water_stress` also calls `irrigated_water_uptake`, `water_uptake_twolayer` and `water_uptake` which are not implemented, but the two first can also be inlined. Whether the third can be could be up for debate.

3 Testing

Since the correctness of the translation is defined as equivalence to its reference, testing is most straightforwardly done by running both the translation and the reference with the same inputs and comparing the outputs. This is done, in practice, by extracting the input and resulting values from a given reference function, and then applying the translation to the same, and comparing results.

This poses several problems. The solution found was to modify the reference implementation with code that generates futhark code using the reference values. For functions that have several exits, this can be done several times, but currently has just been done with the longest path, as most divergent path merely return objects with default values, or somehow signal failures.

This means that only one test is performed per function, which means that we do not necessarily test every path of every sub function. There was one

case where a function failed, but only in a test of a parent function, because in the given test for the function itself, it never took the failing path, which was not taken before the parent function call, and the first function call was from an ealier parent.

We use a mutex macro in the reference implementation to ensure that the test generation is only done once per function, per run. We also do it all in one block, such that test generation is atomic. Otherwise we might risk mixing data from different calls to a given function.

If you want to be able to test multiple execution paths, collecting input and output at the end of these paths seems to be the most elegant approach, both to ensure correctness and for ease of implementation.

When a function takes an object as input (a record in Futhark), there are two basic approaches to generating test code for it. Either one creates a literal record containing all the fields of it, or one uses a constructor which does the same, and then in-place updates the fields relevant for the given test. The relevant fields are those read by the given function, and by all the functions called by this, recursively, i.e. all descendants.

This second method is easier to some extent, but more error prone. One can spent several hours in confusion due to having overlooked a single field, read by a tiny helper function.

3.1 Elements of a test

First, one must find the place in the function, right before it exits, where the input and output values are all known. It is assumed that the input values have not been modified, but this is an oversight. If they are modified in the function, they must additionally be copied and thereby saved, such that their initial state is available at the end.

One then uses the mutex macro and initializes the ostringstream used to write the contents of the testfile:

```
1 if (FIRST_TIME_HERE) {
2   ostringstream oss;
3   init_oss(oss);
4   // rest goes here
5 }
```

The source for the macro is cited in a comment in the code.

Beginning a test

The initializer function does this:

```
void init_oss(std::ostringstream& oss){
  oss << "open import \"../../futsource/everything\"" << endl;
  oss << endl;
  oss << "let input =" << endl;
}
```

which results in the following futhark code:

```
open import "../../futsource/everything"

let input =
```

This is necessary to ensure that all tests import everything they might need. The `everything.fut` file just exists to make test generation not have to worry about which exact libraries are needed. It also begins the input tuple. Since global defintions can not be shadowed, and records that we are going to inplace update must be defined and updated in a a sub expression. This is done with the `input` tuple.

Defining input values

The next step involves generating all the arguments for the function. If for example, the function receivs a simple `real`, one writes:

```
dec_real(oss, "daylength", daylength);
```

Using the function:

```
void dec_real(std::ostringstream& oss,
              const std::string& var,
              const double val
              ){
  oss << "  let " << var << " : " << real_string << " = " << val << endl;
}
```

to generate code like this:

```
  let daylength : f64 = 10.8269
```

If one needs an object, one can declare this in two ways. Either by `init_obj` and `inplace_update`:

```
void init_obj(std::ostringstream& oss,
              const std::string& object,
              const std::string& object_gen
              ) {
  oss << "  let " << object << " = " << object_gen << endl;
}

void inplace_update(std::ostringstream& oss,
                    const std::string& object,
                    const std::string& field,
                    const std::string& value
                    ) {
  oss << "  let " << object << " = " << object << " with " << field << "
↪  = " << value << endl;
}
```

In this manner:

```
init_obj(oss, "gridcell", "Gridcell()");
inplace_update(oss, "gridcell", "lat",
↪  to_string(p.stand.get_gridcell().get_lat()));
```

Resulting in:

```
let gridcell = Gridcell()
let gridcell = gridcell with lat = 19.799999
```

Or, if one requires many updates like this, one can use the function `obj_with_fields`:

```
void obj_with_fields(std::ostringstream& oss,
                    const std::string& object,
                    const std::string& object_gen,
                    const string fields_values[],
                    const unsigned int num_fields
                    ) {
  init_obj(oss, object, object_gen);
  for (unsigned int i = 0; i < (num_fields*2); i+=2){
    inplace_update(oss, object, fields_values[i], fields_values[i+1]);
  }
  oss << endl;
}
```

Like this:

```
string ps_stresses_fields_values[] =
  {"ifnlimvmax", to_string(ps_stresses.get_ifnlimvmax()), // bool.i32
    "moss_ps_limit", to_string(ps_stresses.get_moss_ps_limit()),
    "graminoid_ps_limit",
↪   to_string(ps_stresses.get_graminoid_ps_limit()),
    "inund_stress", to_string(ps_stresses.get_inund_stress())};
obj_with_fields(oss, "ps_stresses", "PhotosynthesisStresses()",
↪   ps_stresses_fields_values, 4);
```

Resulting in:

```
  let ps_stresses = PhotosynthesisStresses()
  let ps_stresses = ps_stresses with ifnlimvmax = false
  let ps_stresses = ps_stresses with moss_ps_limit = 1.000000
  let ps_stresses = ps_stresses with graminoid_ps_limit = 1.000000
  let ps_stresses = ps_stresses with inund_stress = 1.000000
```

Which is essentially just a combination of the other two. It may be better to just call `inplace_update` by hand, however, than using this combined function.

Finishing the input

Once all the input values have been defined and updated as needed, the input tuple is finished:

```
void finish_input(std::ostringstream& oss,
                  const string input){
  oss << "  in " << input << endl;
  oss << endl;
}
```

eg:

```
finish_input(oss, "(ps_env, ps_stresses, pft, lambda, nactive, vm)");
```

```
  in (ps_env, ps_stresses, pft, lambda, nactive, vm)
```

Defining test cases

If the result of the function is just one simple value, it suffices to generate one entry point for the function in this manner:

```
gen_entry_point_test(oss, "respiration", "resp", "resp", "resp",
↪  to_string(resp));
```

```
void gen_entry_point_test(std::ostringstream& oss
                         ,const string function
                         ,const string function_output
                         ,const string testname
                         ,const string output_element
                         ,const string value
                         ) {
  // print test comment
  oss << "-- Autogenerated test of " << function << " output " <<
↪  function_output << " field: " << testname << endl;
  oss << "-- ==" << endl;
  oss << "-- entry: " << testname << "_test" << endl;
  oss << "-- input {}" << endl;
  oss << "-- output { " << value << " }" << endl;
  oss << endl;

  // print entrypoint
  oss << "entry " << testname << "_test" << " =" << endl;
  oss << "  let " << function_output << " = " << function << " input" <<
↪  endl;
  oss << "  in " << output_element << endl;
  oss << endl;
 }
```

Resulting in:

```
-- Autogenerated test of respiration output resp field: resp
-- ==
-- entry: resp_test
-- input {}
-- output { 0.000004 }

entry resp_test =
  let resp = respiration input
  in resp
```

Note that the test technically has no input defined, only an output. Due to the complexity of flattening complicated objects in order to pass them as simple values to futhark standard input, and then reconstruct the records, and since the data is known before the test is generated, is is much simpler to just give no standard input, but to define the value as described above. However, the output must be defined as a standard output. If the test merely did an equality check between the function output and the reference result,

`futhark test` would just report a boolean value, which is of little help in debugging. In this manner, the result is known, and can be used in debugging.

When a function returns multiple results, either as a record with multiple fields, or just a tuple, one test entry must be generated for each of these, to get around the otherwise resultant opacity of the entry points.

```
  void gen_entry_point_tests(std::ostringstream& oss
                             ,const string function
                             ,const string function_output
                             ,const string fields_values[]
                             ,const unsigned int num_fields
                            )
    {
      for (unsigned int i = 0; i < (num_fields*3); i+=3){
        gen_entry_point_test(oss, function, function_output,
↪  fields_values[i], fields_values[i+1], fields_values[i+2]);
      }
    }
```

```
  string res_fields_values[] =
    {"vm", "vm", to_string(vm)
    ,"vmaxnlim", "vmaxnlim", to_string(vmaxnlim)
    ,"nactive_opt", "nactive_opt", to_string(nactive_opt)};
  gen_entry_point_tests(oss, "vmax", "(vm, vmaxnlim, nactive_opt)",
↪  res_fields_values, 3);
```

Giving us these multiple entrypoints:

```
  -- Autogenerated test of vmax output (vm, vmaxnlim, nactive_opt) field: vm
  -- ==
  -- entry: vm_test
  -- input {}
  -- output { 180.282661 }

  entry vm_test =
    let (vm, vmaxnlim, nactive_opt) = vmax input
    in vm

  -- Autogenerated test of vmax output (vm, vmaxnlim, nactive_opt) field: vmaxnlim
  -- ==
  -- entry: vmaxnlim_test
  -- input {}
  -- output { 1.000000 }

  entry vmaxnlim_test =
    let (vm, vmaxnlim, nactive_opt) = vmax input
    in vmaxnlim

  -- Autogenerated test of vmax output (vm, vmaxnlim, nactive_opt) field: nactive_opt
  -- ==
  -- entry: nactive_opt_test
  -- input {}
  -- output { 0.012033 }

  entry nactive_opt_test =
    let (vm, vmaxnlim, nactive_opt) = vmax input
    in nactive_opt
```

subsubsection name

3.2 Complete example

Take for example the `get_co2()` funktion from `canexch.cpp`. This is the reference implementation:

```
double get_co2(Patch& p, Climate& climate, Pft& pft) {
        double pftco2 = climate.co2;
        if (p.stand.is_highlatitude_peatland_stand() && pft.ismoss())
                pftco2 = p.soil.acro_co2; // override for peat mosses
        return pftco2;
}
```

Notice that it only takes three arguments, and how simple the function itself is. However, due to massive use of indirection, the translation becomes:

```
let get_co2(climate : Climate, pft : Pft, stand: Stand, soil : Soil, g:
↪  Gridcell) : real =
  if (is_highlatitude_peatland_stand(stand, g) && ismoss(pft))
  then soil.acro_co2 -- override for peat mosses
  else climate.co2
```

The fields that are accessed in this function, and all the call-descendants, are `stand.landcover, gridcell.lat, pft.lifeform, soil.acro_co2, climate.co2`, so just one field for each object. The `Patch` object falls away from the function, as it is only used to point to another object.

The code necessary to generate a test of this, using the current method, then is the following. We also need to generate dummy `Date` and `Soiltype` objects for use in the constructors, and the result of teh `Stand` constructor has to be handled in a unique way, as it also constructs other objects that are irrelevant to this test.

```
if (FIRST_TIME_HERE) {
  ostringstream oss;
  init_oss(oss);

  init_obj(oss, "date", "Date()");
  init_obj(oss, "soiltype", "Soiltype(0)");

  init_obj(oss, "(_, _, stand)", "Stand(0,0,0,0,0,0,date)");
  inplace_update(oss, "stand", "landcover",
↪  to_string(p.stand.landcover));

  init_obj(oss, "gridcell", "Gridcell()");
  inplace_update(oss, "gridcell", "lat",
↪  to_string(p.stand.get_gridcell().get_lat()));

  init_obj(oss, "pft", "Pft(0)");
  inplace_update(oss, "pft", "lifeform", to_string(pft.lifeform));

  init_obj(oss, "soil", "Soil(soiltype)");
  inplace_update(oss, "soil", "acro_co2", to_string(p.soil.acro_co2));

  init_obj(oss, "climate", "Climate(0.0, 0, 0)");
  inplace_update(oss, "climate", "co2", to_string(climate.co2));

  finish_input(oss, "(climate, pft, stand, soil, gridcell)");

  gen_entry_point_test(oss, "get_co2", "pftco2", "pftco2", "pftco2",
↪  to_string(pftco2));

  gen_test_file(oss, "get_co2");
}
```

And the generated testfile is:

```
open import "../../futsource/everything"

let input =
  let date = Date()
  let soiltype = Soiltype(0)
  let (_, _, stand) = Stand(0,0,0,0,0,0,date)
  let stand = stand with landcover = 4
  let gridcell = Gridcell()
  let gridcell = gridcell with lat = 19.799999

  let pft = Pft(0)
  let pft = pft with lifeform = 1

  let soil = Soil(soiltype)
  let soil = soil with acro_co2 = 934.000000

  let climate = Climate(0.0, 0, 0)
  let climate = climate with co2 = 296.378500

  in (climate, pft, stand, soil, gridcell)

-- Autogenerated test of get_co2 output pftco2 field: pftco2
-- ==
-- entry: pftco2_test
-- input {}
-- output { 296.378500 }

entry pftco2_test =
  let pftco2 = get_co2 input
  in pftco2
```

This is already quite involved for a small function. For the more complex functions, many more objects, with more fields, are required. Further, many functions require arrays of objects. Currently the testing functions are not capable of expressing this, but it could be extended. This can be done by first generating a default record, and then replicating it with the given length of the array, and finally updating the record with the necessary fields, and then the array with the updated record. In case multiple entries are needed, a more complicated approach would be necessary.

It may be simpler to just write one function for each class that generates futhark code for a literal of the class, than to track down each field of each objects, used in each sub function, and write specialized test generation code for each argument object, for each function.

4 How to

In order to run, test and develop further, several dependencies are necessary. In arch or arch derived linux, you will probably need the `cmake` and `netcdf` packages, not to mention futhark iteself. On another distro, there will be some equivalent packages you will need.

The different packages available for futhark in the arch repositories are all several versions behind, so I would recommend installing a precompiled nightly build from the futhark website itself.

Then:

1. Clone the github repository.
2. Add the `input` folder, with its contents, to the `lpj-guess-22/lpj-guess/example/` directory. This is not tracked in the repository due to its size.
3. Notice that the `Makefile` is located in `lpj-guess-22/`.
4. `Make 0` to typecheck the futhark code files.
5. `Make 1` to compile the reference implementation.
6. `Make 2` to run the reference implementation, and thereby generate testfiles for the futhark implementation.
7. `Make 3` to run the testfiles for the futhark implementation.

5 Current state and a few recommendations

Currently there is a use-after-consume[4] bug in the implementation of the `canopy_exchange()` function. Normally these can be fixed using `copy`, but this does not seem to be the case.

The code relies on some global constants that are set in the parameter files, and in the `global.ins` file.

In several places in the code, `TODO`, `FIXME` etc are used to denote this and other issues. A couple of functions have yet to be implemented for `canopy_exchange` to work, specifically `irrigated_water_uptake`, `water_uptake_twolayer` and `water_uptake`. The first of these has a few dependencies of its own in the soil module.

I would also recommend writing functions for the testing framework that just print literals of entire objects. This will make it much more straightforward to later on begin testing arrays of objects, and it will be less error prone, as one will no longer have to keep track of data dependencies of subfunctions of subfunctions, although writing these print functions will require some investment up front.

After finishing `canopy_exchange` with its dependencies, one can move on to the other functions in the main outer loops.

References

[1] B. Smith, "Lpj-guess-an ecosystem modelling framework," *Department of Physical Geography and Ecosystems Analysis, INES, Sölvegatan*, vol. 12, p. 22 362, 2001.

[2] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, Barcelona, Spain: ACM, 2017, pp. 556–571, ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062354.

[3] T. Henriksen, *Giving programmers what they want, or what they ask for*, Jan. 2019. [Online]. Available: https://futhark-lang.org/blog/2019-01-13-giving-programmers-what-they-want.html#.

[4] *Compiler error index*. [Online]. Available: https://futhark.readthedocs.io/en/latest/error-index.html#use-after-consume.