SoftwareX 19 (2022) 101200

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

EspyInsideFunction.jl — extracting intermediate results from numerical functions

Philippe Mainçon

SINTEF Ocean, Postboks 4760 Torgarden, 7465 Trondheim, Norway

ARTICLE INFO

Article history:
Received 4 February 2022
Received in revised form 8 August 2022
Accepted 25 August 2022

Keywords:
Meta-programming
Julia language
Finite element analysis

ABSTRACT

`EspyInsideFunction` allows to write software in the Julia programming language Julia (2017) [1] to make the value of variables within a function's local scope — variables that are neither arguments nor return values, available to the caller. This is relevant for functions within a solution process (e.g. a function which return value is to be minimized by some iterative scheme). In such a setting it is natural to tailor the function's interface to the solution process. However, internal results within the function, while not relevant to the solution process, may be wanted output from the analysis. The package allows to write such a function with an interface tailored for the solution process, and then uses meta-programming to create a second version of the function, with a modified interface, which can be called to extract relevant intermediate results.

Code metadata

Current code version	0.2.1
Repository	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00032
Code Ocean compute capsule	n/a
Legal Code License	MIT License
Compilation requirements, operating environments & dependencies	Julia 1.7+
Documentation/manual	philippemaincon.github.io/EspyInsideFunction.jl
Support email for questions	philippe.maincon@sintef.no

1. Motivation and significance

In the process of computing return values from their arguments, functions typically assign intermediate results to local scope variables. These intermediate results are not returned by the function because they are not of interest for the code calling the function. Consider a finite element code: Each finite element type has a function (or method) `residual`, whose arguments include degrees of freedom (for example: displacements), and whose return values include residuals (in the same example: forces). The solution process then adjusts the degrees of freedom in order to make all residuals (adequately summed over all elements in the model) close to zero (force equilibrium).

The intermediate results (for example stresses) are not needed *by* the solution process: the solution algorithm does not explicitly use stresses to adjust the degrees of freedom. However some intermediate results may be wanted *from* the solution process: a typical motivation for a finite element analysis of a structure is to determine the stresses within the structure, so stresses need to be available for visualization and other post processing.

One obvious solution would be to modify the function interface, and return any intermediate results wanted from the solution process in addition to those wanted by the solution process. This approach has its drawbacks:

1. Not all intermediate results may be wanted from the solution process, but the quantity of intermediate results of *potential* interest may be large. To avoid unnecessary storage of intermediate results, the user would have to specify which results are wanted. The code of the function would be cluttered by test and storage code.
2. Handling the transfer of intermediate results also clutters the code of the solution process.

E-mail address: philippe.maincon@sintef.no.

https://doi.org/10.1016/j.softx.2022.101200

EspyInsideFunction was developed as part of a project in which a in-house finite element code (referred to thereafter as "Lithe") was written in the Julia language. The emphasis in this project was on making it as easy as possible to implement new types of finite elements and new solvers, so that modifying the functions interface was not preferred.

2. Software description

2.1. Approach

EspyInsideFunction 's approach is to use Julia's meta-programming functionality to generate two versions of the "espied" function's code (in the above example: `residual`):

1. The fast version, which is essentially the "espied" function un-tampered, and does nothing to save or export intermediate results. This version is used by the solution process.
2. The "spying" version, which is called to extract intermediate results. It receives additional parameters:
   - `key`, (input) which describes what intermediate results to store, and where to write them into the vector `out`.
   - `out`, (pre-allocated output) a vector (or vector-shape view into a larger array) in which to store intermediate results.

   Code is inserted in this version to store intermediate results where requested.

The numerical analysis uses EspyInsideFunction as follows:

1. The solution process runs using the fast version and the solver stores the inputs to the espied function (in the example, degrees of freedom and internal states, for each finite element).
2. Then, the spying version of the code can be called with said inputs, and a request for intermediate results, to extract the intermediate results.

This design provides great flexibility. For example, a thread could be used to extract and present results as the analysis progresses, with the possibility to adjust on the fly which results are exposed. Another possibility is to store all the inputs to the espied function, and to provide functionality to allow the user to interactively request intermediate results. In both cases: there is no need to decide before the analysis which intermediate results are of interest.

Another important aspect of EspyInsideFunction 's design is memory management: this "espying" approach can be used within loops over multiple function calls (in the example: on for each element in a finite element mesh, and for each load increment in the analysis). To avoid memory fragmentation and garbage collection overhead, all intermediate results are copied to a single large array. This requires a `key`, discussed in the following, to address elements within the array. More specifically, the `key` provides indices into "flattened" vector of intermediate results extracted from *one* call to the espied function. For multiple calls, the array of results is created with additional dimensions (in the example, element and increment).

2.2. Software functionalities

The developer of a function to be espied must provide said function, annotated to point out available intermediate results, and a data structure `requestable`, which provides a description of the name and size of available intermediate results, including the handling of intermediate results that appear inside of loops or functions called by the espied function. See Section 3.2 for examples of such functions.

EspyInsideFunction provides three components: `@request` to specify which intermediate results are wanted, `makekey` to generate the above-mentioned `key` and `@espy` to generate "fast" and "spying" code for the espied function.

@request. The macro `@request` allows the user to describe which intermediate results are wanted. For example

```
req = @request gp[].(s,material.(a,b))
```

states that withing the espied function, there is a loop of fixed length of the form

```
for igp = 1:ngp
```

and inside the loop, the variable `s` is requested. Within the same loop, there is also a call to a function called `material`, within which variables `a` and `b` are requested.

The macro just captures the string and returns the syntax tree for that string, as generated by Julia's parser.

Makekey. The function `makekey` takes as input the syntax tree generated by `@request` and the data structure `requestable`. Combining both, it returns the `key`. This `key` is a data structure of nested vectors and named tuples, with names corresponding to names in the espied function: sub-functions, loop variables and variable names. The content of this data structure is indices into a vector `out` that will be returned by the "spying" version.

@espy. This is the macro, which, given annotated code, generates the fast and spying codes. A variant `@espydbg` is also provided, that outputs both codes.

2.3. Software architecture

The generation of the spying code is made in several passes: `@espy` generates the fast code, and a precursor to the spying code. This precursor code contains further macro calls: `@espy_loop`, placed at the top of for-loop blocks, `@espy_record`, placed right after an assignment to an espied variable and `@espy_call`, placed right after a call to an espied sub-function.

This precursor code can be inspected by replacing the `@espy` macro invocation by the verbose `@espydbg`.

In the present version, `@espy_loop`, `@espy_record`, `@espy_call` generate a small code with a test on what is requested in `key`. One could get some performance improvement by having these macros generate different code depending on the `key`.

3. Illustrative examples

3.1. Package documentation

In addition to the example below, the documentation provides a complete usage example: `github.com/PhilippeMaincon/EspyInsideFunction.jl/ blob/master/test/EspyDemo.jl`.

3.2. Basic usage

The following shows annotated code for espied function `residual` and a sub-function `material`. Note the use of colons to annotate variables that receive an assignment, the annotated function call, and the `@espy` macros at the head of both function declarations. The data structure `requestable` (used by

makekey) here states that within a for loop of length 2, two scalars z and s are requestable, and that within the espied function, there will be a call to the function material, within which more results are requestable.

```
using EspyInsideFunction
@espy function residual(x, y)
    ngp = 2
    r   = 0
    for igp  = 1:ngp
        :z    = x[igp]+y[igp]
        :s, r = :material(z)
    end
    return r
end
@espy function material(z)
    :a = z+1
    :b = a*z
    return b, 3.
end
requestable = (gp = forloop(2, (z = scalar, s = scalar,
material    = (a = scalar, b = scalar))),)
```

The following is the code written by the user to extract intermediate results.

```
req        = @request gp[].(s, material.(a, b))
key, nkey = makekey(req, requestable)
out        = Vector{Float64}(undef, nkey)
x, y       = [1.,2.], [.5,.2]
r          = residual(out, key, x, y)
b2         = out[key.gp[2].material.b]
```

The user first specifies which results are wanted, in a custom syntax.

A key is generated which allows the spying function to know where to write intermediate results, and the user know how to access these. In the present example, key is such that

```
key.gp[1].s          == 1
key.gp[1].material.a == 2
key.gp[1].material.b == 3
key.gp[2].s          == 4
key.gp[2].material.a == 5
key.gp[2].material.b == 6
```

Note that the integers in key are unique, and form a sequence. nkey is the value of the highest integer, useful for allocating the result storage out. If a was an array, then key.gp[1].material.a would be an array of integers, of the same shape as a.

An array out is then allocated to store the intermediate results.

Assuming the solution process is completed (values of x and y have been obtained, using the fast version of residual), the spying version of residual is called. Note the two additional parameters.

The key is finally used to address results of interest in out.

3.3. Integration into a finite element software

In the finite element software Lithe which motivated the present package, a data type is associated to each finite element type (volume, point load, beam etc.). Methods residual (to be espied) and requestable are associated to each element data type. An element "has-a" and calls, a material model, and materials are organized in a similar fashion:

```
struct Volume
    N :: Matrix{Float64}
    ...
end

@espy function residual(o::Volume, y)
    ... # interpolations, call to material model, quadrature
    return r
end

function requestable(o::Volume)
    at_gp = (F = (Nx, Nx), material = requestable(o.mat))
    return (X = (Nnod, Nx), gp = forloop(Ngp, at_gp))
end

@espy function material(o::Material, F)
    ...
    return stress
end

function requestable(o::Material)
    return (strain = (Nx, Nx), stress = (Nx, Nx))
end
```

Then the user writes a script to run the analysis and extract the results:

```
... # code for meshing and boundary conditions
state    = run(analysis) # state[istep][iincrement].y
                         # are degrees of freedom

req      = @request X, gp[].(F, material.(stress, strain))
out, key = elementresult(state[istep], req, Volume,
               iels = 1:1000, iincrs=[2,3,50])
strain   = out[key.gp[2].material.strain, ielement, iincrement]
```

The finite element solver returns a data structure state with the "essential" results of the whole analysis. These are the degrees of freedom of the system (and, taken out of the above example for clarity, the state variables for each element, for example plastic strains).

Lithe provides the method elementresult that can be used to extract intermediate results from multiple elements of the same type, at multiple load increments. Building on this, Lithe also provides functionality to extract intermediate results, interpolate them to the model's nodes and create a visualization. This is not described further here.

Using dispatching, elementresults calls the relevant requestable method, and uses the output to call makekey. It then has the necessary information (nkey) to allocate a multi-dimensional array out in which to store the intermediate results, for all requested elements and load increments. elementresults then loops over increments and elements. Within the loop, it calls the spying version of the relevant residual method with each element's internal state and degrees of freedom, and a slice of out.

4. Impact

In Lithe, the use of EspyInsideFunction was combined with automatic differentiation [2,3] to simplify the code of residual: The following is from the implementation of a simple displacement-based element for static problems in mechanics.

```
@espy function residual(o::Volume, t::Real, y, r::Real1,
                        stateo, staten, statecv)
    ixdof = o.kind.ixdof
    :X    = y[ixdof].+o.Xo
    for igp = 1:ngp
        N, B, dVo  = o.kind.N[igp], o.B[igp], o.dVo[igp]
```

```
        :F          = (B*X0)'
        S, f, staten[igp] = :material(o.mat, t, F,
                        stateo[igp], statecv)
        r[ixdof'] += (f*N' + F*S*DN)*dVo
    end
end
```

Without any mention of incremental matrices, and no visible code to extract intermediate results, the numerical formulation of the element can be quickly read from the code. Compared to a previous generation of code in Fortran 90, written without automatic differentiation and automated extraction of intermediate results, the length of code is reduced by a factor 10 for simpler elements and more for hybrid formulations, without loss of performance.

At the same time, the solver code in Lithe (vector and matrix assembly) is written without concern for result extraction, there also improving readability.

Within Lithe, `EspyInsideFunction` has contributed to lowering the cost and complexity of creating and maintaining high performance finite element solutions. These lower costs will facilitate the creation of commercially viable FEM solvers for niche applications (like [4,5], which were written without `EspyInsideFunction`), and other tailor-made solvers by researchers. By taking out the drudge part of element development, this also lowers the threshold to induce students into this type of work.

The technique is expected to be applicable other numerical software, in particular those with

- iterative solutions
- plug-in components that should have simple code

including, computational fluid dynamic code, neural networks, and so forth. It must be noted that the macro `@espy` was developed specifically for FEM applications, and thus may need some extension to tackle Julia syntax not used in the FEM context.

Julia language's macros, operating on syntax trees inspired and allowed the creation of `EspyInsideFunction`.

5. Conclusions

`EspyInsideFunction` demonstrates that it is possible to extract intermediate results from within functions written in the Julia programming language, without code clutter and with good performance. `EspyInsideFunction` allows to create numerical procedures in which the exact results wanted from the analysis can be decided during *or after* the analysis.

There is of course scope for improvement:

- It should not be necessary to provide a `requestable` data structure. `@espy` could generate a third version of the espied call, which when called (only once) would survey the name and size of each intermediate result.
- More general Julia syntax should be supported. For example, loops with `enumerate` are not supported in the present version.
- Bugs in the espied functions lead to error messages which are not made less readable by `@espy`. However internal errors in `@espy` (failing to process unsupported Julia syntax) or in the spying version of the espied function (disagreement on array size) result in arcane error messages.
- A less terse and rigid syntax for inputs to `@request` might be easier to use.
- The type of the output of function `makekey` depends on the input. This in turn means that the spying version of the code will be recompiled for "each" request. Ensuring `makekey` is type-stable would improve performance.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

`EspyInsideFunction` was developed as a part of the Lithe project, awarded by SINTEF Industry, financed by the Research Council of Norway's general subvention to SINTEF Industry.

Several anonymous users on `discourse.julialang.org` generously contributed their time and expertise in helping this author understand basic and advanced features of Julia needed to create `EspyInsideFunction`. The reviewer to this paper provided extensive feedback which significantly contributed to the quality of both this paper and the software. These contributions to `EspyInsideFunction` are gratefully acknowledged.

This application was inspired by the capabilities of the Julia programming language. The author expresses admiration and gratitude to Julia's creators.

Current executable software version

At the Julia prompt, type

```
julia> using Pkg
julia> Pkg.add("EspyInsideFunction")
```

to download and install the latest version of the package (including dependencies) from the Julia registry. The package should then be ready for use:

```
julia> using EspyInsideFunction
```

References

[1] Bezanson Jeff, Edelman Alan, Karpinski Stefan, Shah Viral B. Julia: A fresh approach to numerical computing. SIAM Rev 2017;59:65–98. http://dx.doi.org/10.1137/141000671.

[2] Wengert RE. A simple automatic derivative evaluation program. Comm ACM 1964;7(8):463–4. http://dx.doi.org/10.1145/355586.364791.S2CID24039274.

[3] Revels J, Lubin M, Papamarkou T. Forward-mode automatic differentiation in Julia. 2016, arXiv:1607.07892.

[4] Bruaseth Sævik. Theoretical and experimental studies of the axisymmetric behaviour of complex umbilical cross-sections. Appl Ocean Res 2005;27(2):97–106, 2005.

[5] Hoang Hieu, Mainçon Philippe, Philippe David, Coudert Terence, Bjørset Arve, Saether Sturla. Novel computational tool for efficient structural analyses of geothermal wells. Geothermics 2021;92:102058.