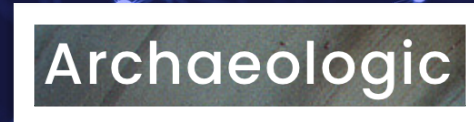


Fortran at the Intersection:

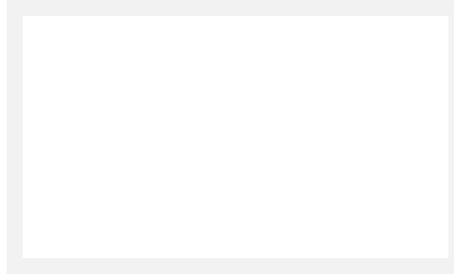
Synergies Arising from the Interplay Between Paradigms

Damian Rouson

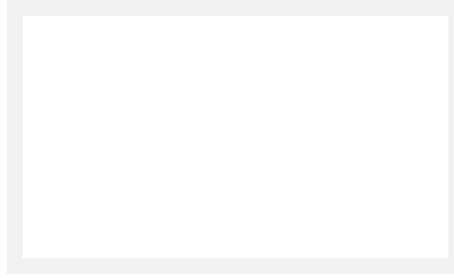


Outline

- Introduction
- Case Studies
- Intersectionality
- Acknowledgments



Outline



Introduction

- ⌘ Background: Fortran at the Intersection.
- ⌘ Motivation: A preliminary summary.



Case Studies



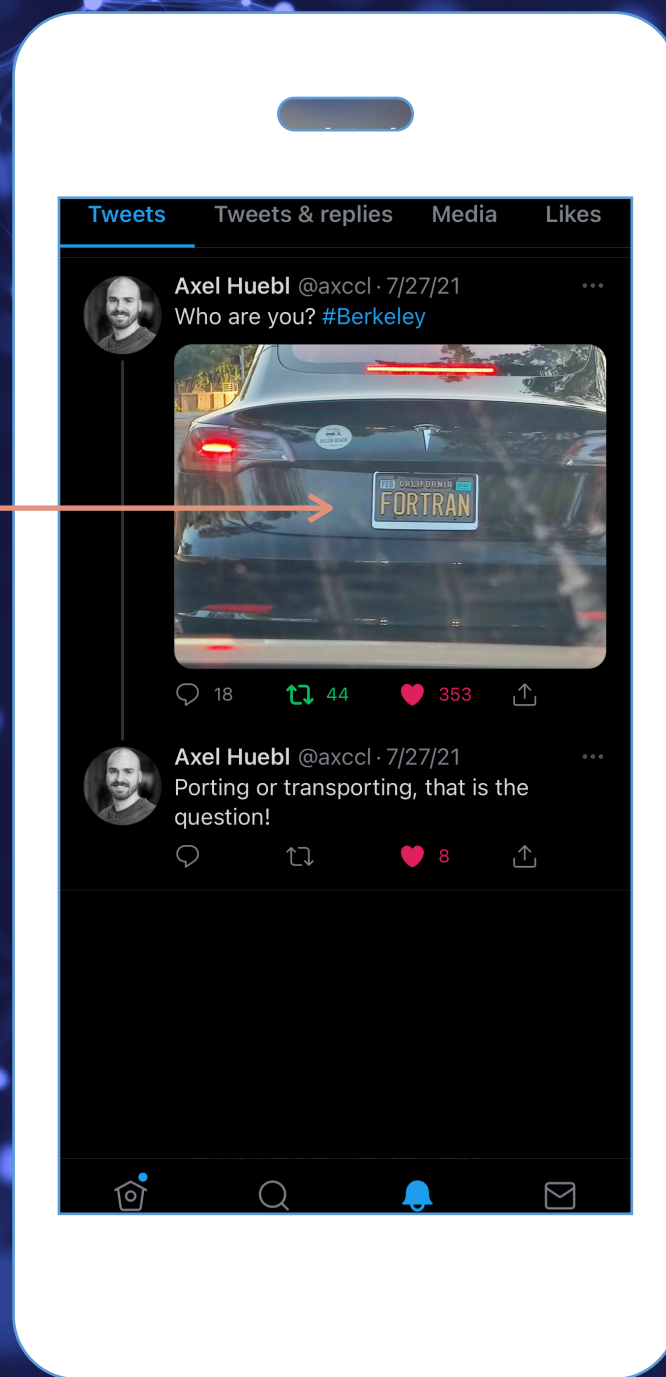
Intersectionality

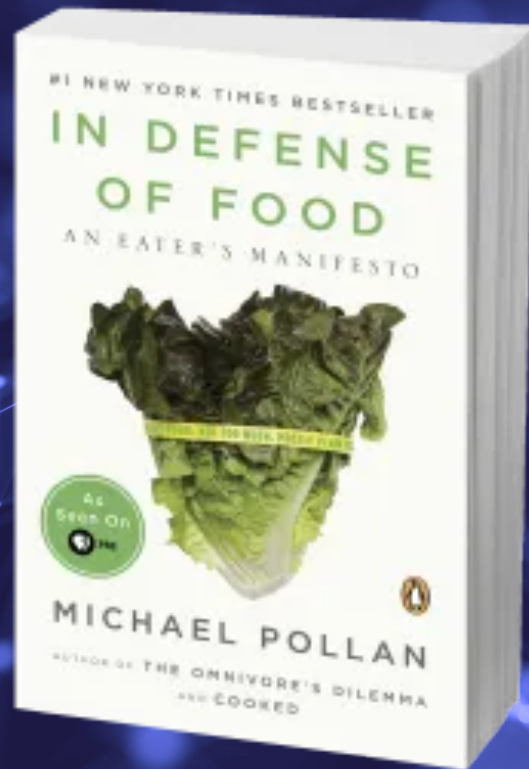


Acknowledgments



Fortran at the Intersection






Eat food.

Not too much.

Mostly plants.



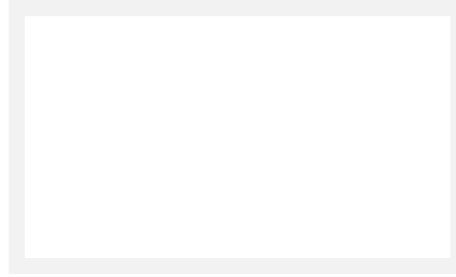
In Defense of Software: A Developer's Manifesto

Write software.

Not too much.

Mostly pure functions.

Outline



Introduction

Case Studies: Open-Source Software

- ✱ Rocket Science
- ✱ Synergies at the Intersection of Paradigms
- ✱ FEATS, Dag, and Assert

Intersectionality

Acknowledgments



Rocket Science

An educational mini-app for predicting solid rocket motor thrust.

Instructions to client:

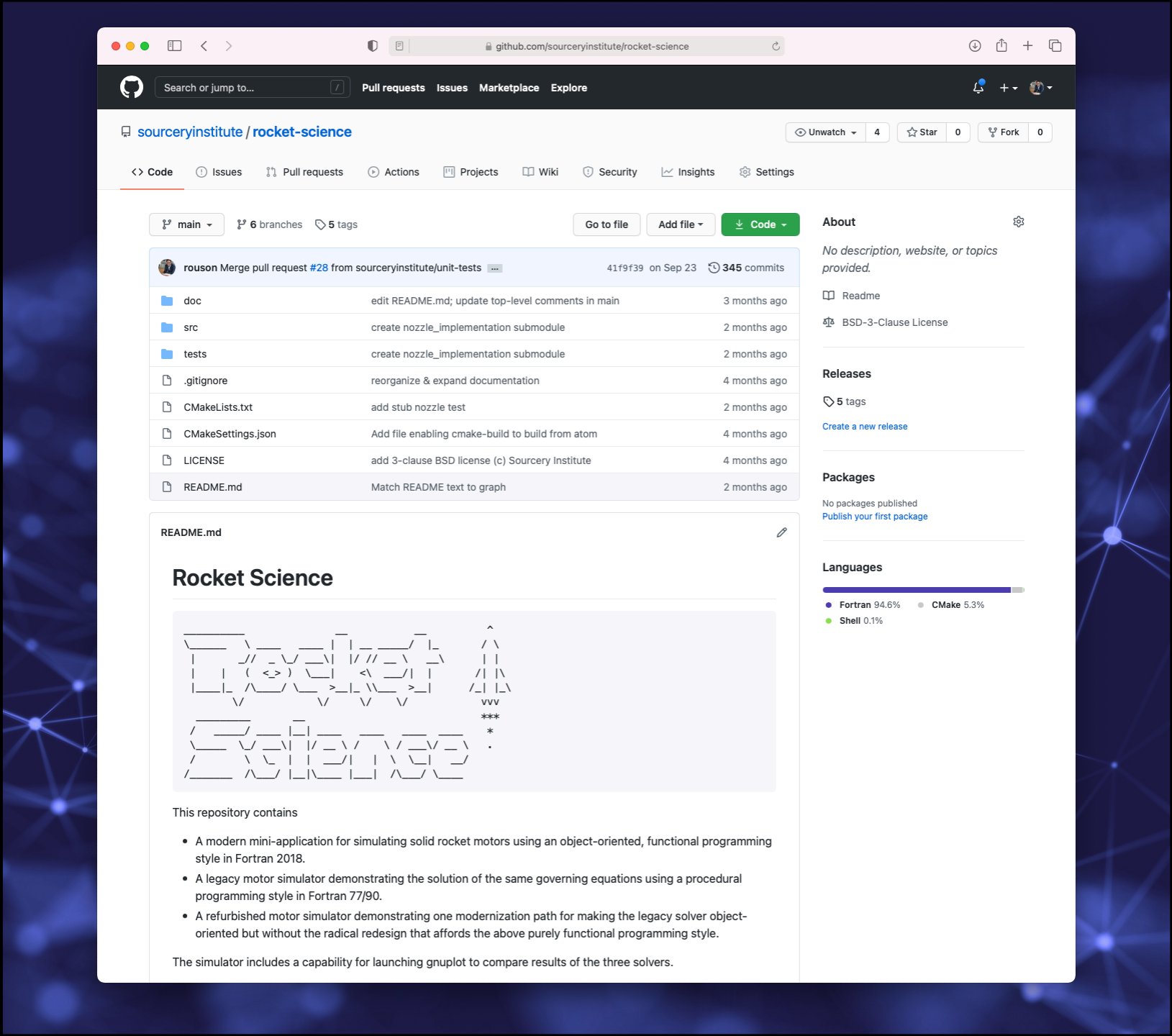
- 🦋 “Write a small proxy application capturing an algorithm that plays an important role in your production application.”
- 🦋 “Write the proxy app exactly as you would the day before you met us.”

Result:

- 🦋 A procedural Fortran 90 code in which subroutines with 0-1 arguments collectively use one common module containing approximately 40 public module variables.

Goals:

- 🦋 To demonstrate an evolutionary approach for refactoring the proxy app using modern programming paradigms.
- 🦋 To use the process as a case study in a training course.



Programming Paradigms

Functional
Programming

Object-Oriented
Programming

Parallel
Programming

Programming
by contract

What is a pattern in one language might be a feature in another language, including a future version of the same language.

Thus, patterns might be strong candidates for future inclusion in the language: some aspects of the current considerations for generic programming resemble programming by contract.

A Functional Programming Pattern

The Pattern:

- ✧ Write **pure**, including **elemental**, functions.
- ✧ **Associate** names with expressions, including function references.
- ✧ Construct whole objects with generic interfaces serving as user-defined structure constructors → **no setters**.
- ✧ Write expressions based on pure user-defined operators.

```
src — vim main.f90 — 135x33
1 program main
2   !! A mini-application for rocket motor simulation:
3   !! demonstrating an object-oriented, functional programming style in Fortran 2018
4   !! with automatic graphing of results in gnuplot for comparison against a legacy,
5   !! procedural simulator written in Fortran 90.
6   use motor_interface, only : motor_t
7   use state_interface, only : state_t
8   use kind_parameters, only : rkind
9   implicit none
10  real(rkind), parameter :: zero=0._rkind
11  character(len=*), parameter :: input_file="rocket.inp"
12
13  type(motor_t) motor
14  type(state_t) state !! state variables updated at each time step: mass, energy, time, and burn depth
15  type(state_t), allocatable :: history(:)
16
17  call motor%define(input_file)
18
19  associate(chamber => motor%chamber())
20    associate(gas => chamber%gas())
21      call state%define(input_file, R_gas=gas%R_gas(), c_v=gas%c_v(), volume=chamber%initial_volume(), time=zero, burn_depth=zero)
22    end associate
23  end associate
24
25  associate(dt => motor%dt() )
26    history = [state]
27    do while(state%time() < motor%t_max())
28      state = state + motor%d_dt(state)*dt
29      history = [history, state]
30    end do
31  end associate
```


A Functional Programming Pattern

The Benefits:

- ✧ Clarity: the reader sees inputs map to outputs without reading the interface.
- ✧ Robustness*: associating with expressions provides immutable state with clear, limited scoping.
- ✧ Performance: only pure procedures may be invoked inside **do concurrent** blocks, easier to write asynchronous parallel code, + other potential optimizations.

* This reduces or eliminates whole categories of errors that arise from the traditional declare-and-define pattern, e.g., no more mistaken use of uninitialized data or mistakenly overwriting data.

```
src — vim main.f90 — 135x33
1 program main
2   !! A mini-application for rocket motor simulation:
3   !! demonstrating an object-oriented, functional programming style in Fortran 2018
4   !! with automatic graphing of results in gnuplot for comparison against a legacy,
5   !! procedural simulator written in Fortran 90.
6   use motor_interface, only : motor_t
7   use state_interface, only : state_t
8   use kind_parameters, only : rkind
9   implicit none
10  real(rkind), parameter :: zero=0._rkind
11  character(len=*), parameter :: input_file="rocket.inp"
12
13  type(motor_t) motor
14  type(state_t) state !! state variables updated at each time step: mass, energy, time, and burn depth
15  type(state_t), allocatable :: history(:)
16
17  call motor%define(input_file)
18
19  associate(chamber => motor%chamber())
20    associate(gas => chamber%gas())
21      call state%define(input_file, R_gas=gas%R_gas(), c_v=gas%c_v(), volume=chamber%initial_volume(), time=zero, burn_depth=zero)
22    end associate
23  end associate
24
25  associate(dt => motor%dt() )
26    history = [state]
27    do while(state%time() < motor%t_max())
28      state = state + motor%d_dt(state)*dt
29      history = [history, state]
30    end do
31  end associate
```

A Functional Programming Pattern

The Roadblocks:

- 🦋 Functions can only have one result.
- 🦋 Producing output for debugging is cumbersome.
- 🦋 If copies replace mutations, memory could be used inefficiently.

```
1  program main
2  !! A mini-application for rocket motor simulation:
3  !! demonstrating an object-oriented, functional programming style in Fortran 2018
4  !! with automatic graphing of results in gnuplot for comparison against a legacy,
5  !! procedural simulator written in Fortran 90.
6  use motor_interface, only : motor_t
7  use state_interface, only : state_t
8  use kind_parameters, only : rkind
9  implicit none
10
11  real(rkind), parameter :: zero=0._rkind
12  character(len=*), parameter :: input_file="rocket.inp"
13
14  type(motor_t) motor
15  type(state_t) state !! state variables updated at each time step: mass, energy, time, and burn depth
16  type(state_t), allocatable :: history(:)
17
18  call motor%define(input_file)
19
20  associate(chamber => motor%chamber())
21    associate(gas => chamber%gas())
22      call state%define(input_file, R_gas=gas%R_gas(), c_v=gas%c_v(), volume=chamber%initial_volume(), time=zero, burn_depth=zero)
23    end associate
24  end associate
25
26  associate(dt => motor%dt() )
27    history = [state]
28    do while(state%time() < motor%t_max())
29      state = state + motor%d_dt(state)*dt
30      history = [history, state]
31    end do
32  end associate
```


Rocket Science

An educational mini-app for predicting solid rocket motor thrust.

Steps to go from software archaeology to modernity:

- 🔧 Set up a git repository.
- 🔧 Add an fpm manifest.
- 🔧 Establish a reference implementation:
 - 🔧 main → function
- 🔧 Set up a test suite using Vegetables.
- 🔧 Define GitHub Actions for
 - 🔧 Automating continuous integration testing.
 - 🔧 Generating ford documentation.
 - 🔧 Deploying documentation to GitHub Pages.
- 🔧 Convert module variables to dummy arguments with intent.
- 🔧 Separate procedure interface bodies (in modules) from procedure definitions (in submodules).
- 🔧 Convert subroutines to pure functions:
 - 🔧 **intent(in)**: no change.
 - 🔧 **intent(out)**: encapsulate in a derived type result.
 - 🔧 **intent(inout)** → intent(in) + result component.

The screenshot shows the GitHub repository page for 'sourceryinstitute/rocket-science'. The repository has 419f39 as the latest commit on Sep 23, with 345 commits in total. It features a file browser with folders 'doc', 'src', and 'tests', and files '.gitignore', 'CMakeLists.txt', 'CMakeSettings.json', 'LICENSE', and 'README.md'. The 'README.md' file is expanded, showing the title 'Rocket Science' and a diagram of a rocket engine. Below the diagram, it states 'This repository contains' and lists three bullet points: a modern mini-application for simulating solid rocket motors, a legacy motor simulator, and a refurbished motor simulator. The repository also includes a capability for launching gnuplot.

Repository: [sourceryinstitute/rocket-science](#)

419f39 on Sep 23 345 commits

File	Description	Time
doc	edit README.md; update top-level comments in main	3 months ago
src	create nozzle_implementation submodule	2 months ago
tests	create nozzle_implementation submodule	2 months ago
.gitignore	reorganize & expand documentation	4 months ago
CMakeLists.txt	add stub nozzle test	2 months ago
CMakeSettings.json	Add file enabling cmake-build to build from atom	4 months ago
LICENSE	add 3-clause BSD license (c) Sourcery Institute	4 months ago
README.md	Match README text to graph	2 months ago

Rocket Science

This repository contains

- A modern mini-application for simulating solid rocket motors using an object-oriented, functional programming style in Fortran 2018.
- A legacy motor simulator demonstrating the solution of the same governing equations using a procedural programming style in Fortran 77/90.
- A refurbished motor simulator demonstrating one modernization path for making the legacy solver object-oriented but without the radical redesign that affords the above purely functional programming style.

The simulator includes a capability for launching gnuplot to compare results of the three solvers.

About

No description, website, or topics provided.

Releases

5 tags

[Create a new release](#)

Packages

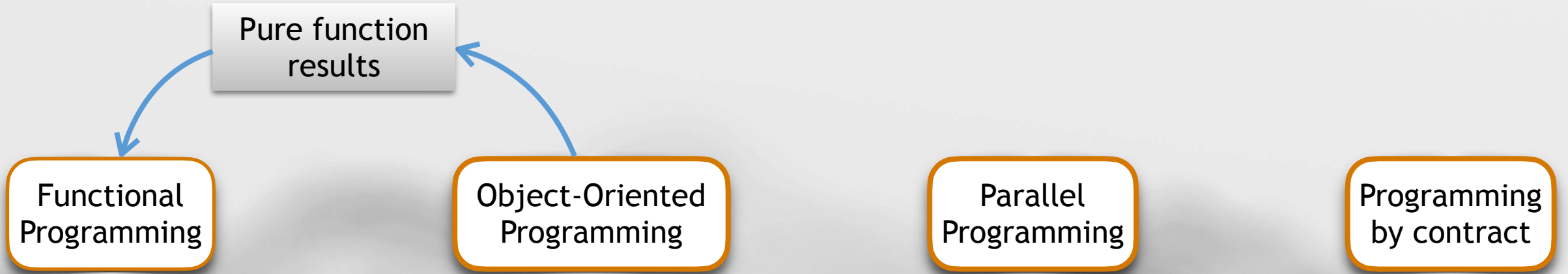
No packages published

[Publish your first package](#)

Languages

Language	Percentage
Fortran	94.6%
CMake	5.3%
Shell	0.1%

Synergies at the Intersection



Patterns resolve tensions in program design:

Using objects as function results resolves the tension between needing multiple results but only being allowed one.

This approach often also leads to a natural encapsulation strategy: if one procedure produces multiple values, it's likely those values are related in a way that makes a reasonable choice of abstractions.

FEATS

Framework for Extensible Asynchronous Task Scheduling

Execution:

- 🧙 In each team, establish one scheduler image and one or more compute images.
- 🧙 Schedulers post `task_assigned` events to compute images in an order that respects dependencies in a directed acyclic graph (DAG).
- 🧙 Compute images post `ready_for_next_task` events to scheduler.
- 🧙 A `task_payload_map_t` abstraction maps task identifiers to locations in a `payload_t` mailbox coarray.

Initial target applications:

- 🧙 NASA's Online Tool for the Assessment of Radiation in Space (OLTARIS)
- 🧙 NCAR's Intermediate Complexity Atmospheric Research (ICAR) model: work-sharing/work-stealing.
- 🧙 Fortran Package Manager: parallel builds.

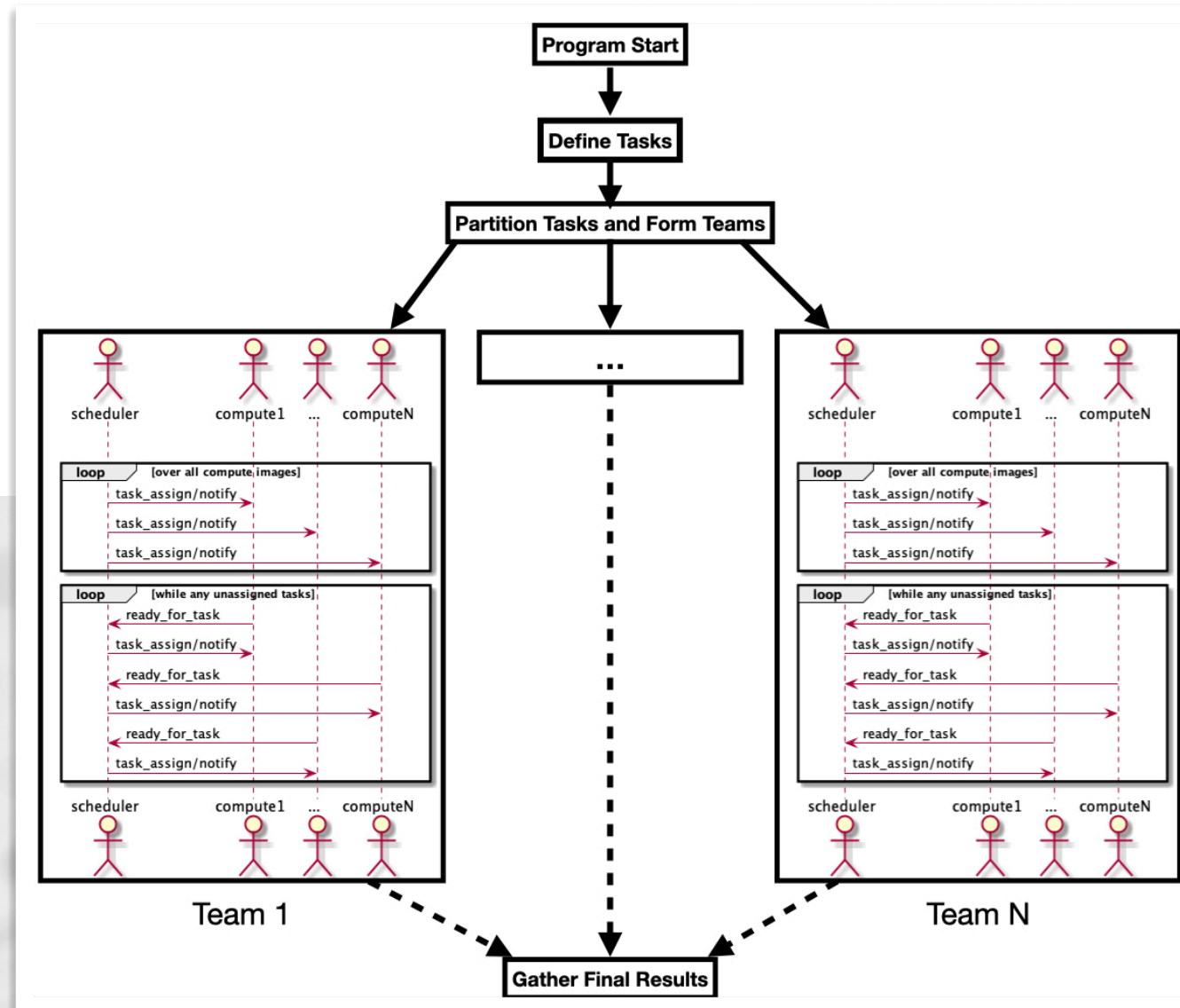
The screenshot shows the GitHub repository page for `sourceryinstitute/FEATS`. The repository is public and has 6 stars, 1 fork, and 2 issues. The main branch is selected, showing 8 branches and 2 tags. The repository contains a table of files and folders with their commit history:

File/Folder	Commit Message	Commit Time
<code>.github/workflows</code>	doc: add and deploy ford documentation	yesterday
<code>doc</code>	doc: add and deploy ford documentation	yesterday
<code>example</code>	doc(example): add README.md in example dir	yesterday
<code>src</code>	feat: replace sourcery dependency with assert	8 hours ago
<code>tests</code>	- Removed data_location_map, since it has becom...	15 days ago
<code>.gitignore</code>	doc: add and deploy ford documentation	yesterday
<code>LICENSE</code>	Initial commit	7 months ago
<code>README.md</code>	doc(README): delete broken link	yesterday
<code>fpm.toml</code>	build(fpm): update dag dependency	8 hours ago

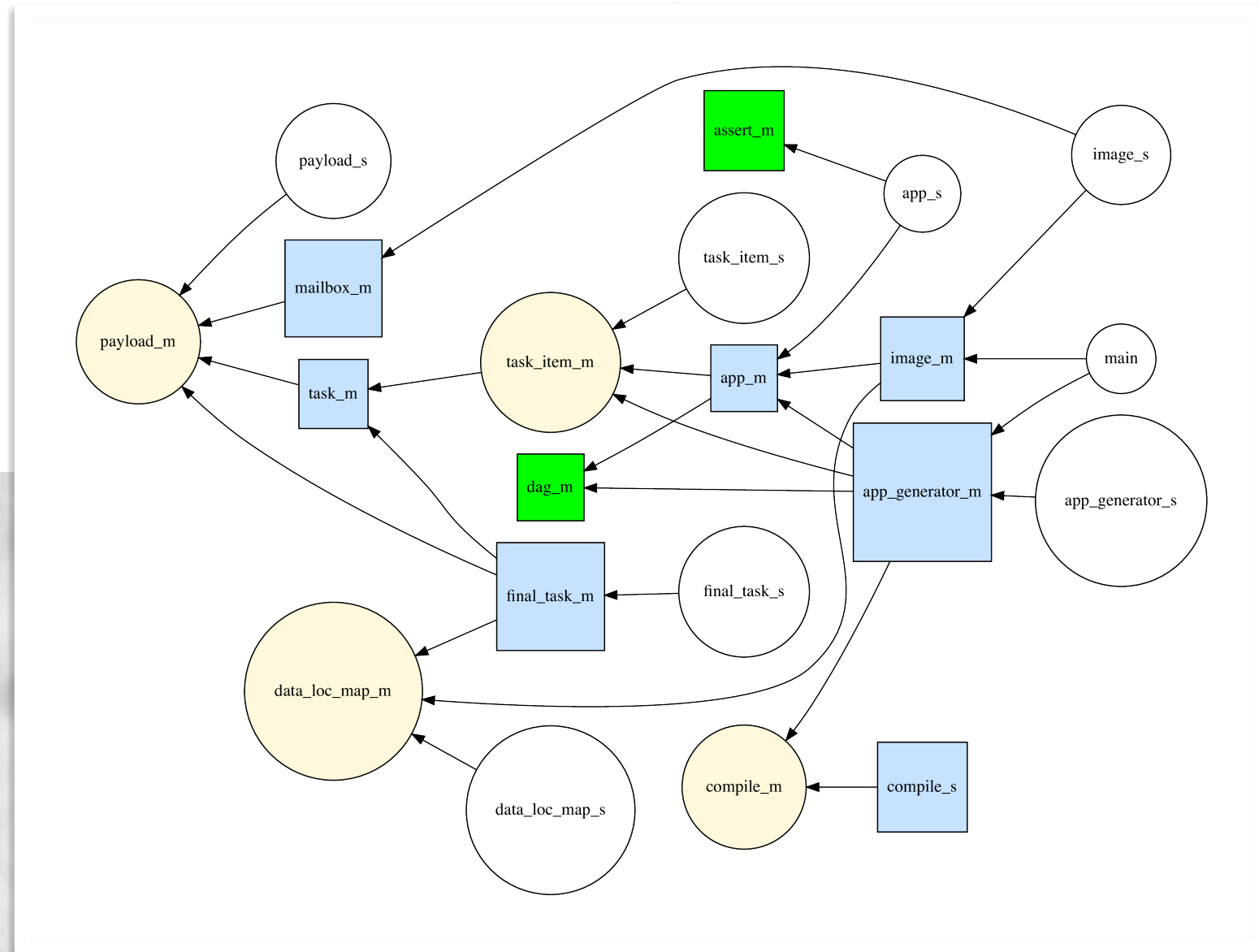
The repository also features a README.md file, which is displayed below the file list. The README includes the Sourcery Institute logo, the project title "Framework for Extensible Asynchronous Task Scheduling (FEATS)", and links to the license (BSD-3), release (v0.2.0), and PDF README. The repository is licensed under BSD-3 and has a release of v0.2.0. The README also includes links to the Overview, Getting Started, Documentation, Dependencies, Acknowledgments, and Donate pages.

The right sidebar of the repository page shows the "About" section, which includes the project description "Framework for Extensible Asynchronous Task Scheduling", the license "BSD-3-Clause License", and the "Releases" section, which shows the latest release "Update to dag ve..." from 7 hours ago. The "Packages" section shows no packages published, and the "Contributors" section lists four contributors: rouson, everythingfunctional, hsnyder, and singleterry. The "Environments" section shows one environment "github-pages" which is active.

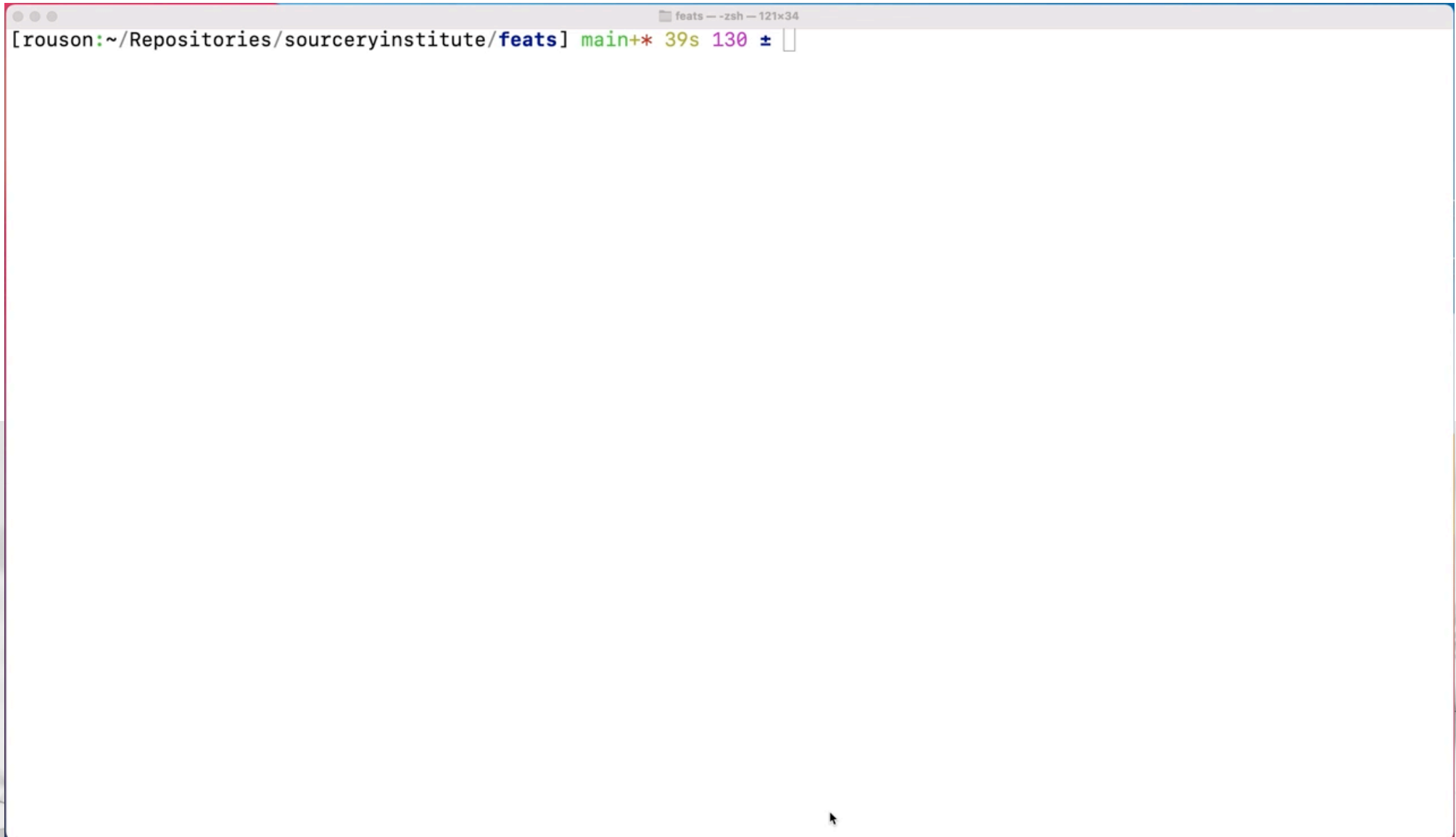
UML Sequence Diagram



FEATS Module Dependency DAG



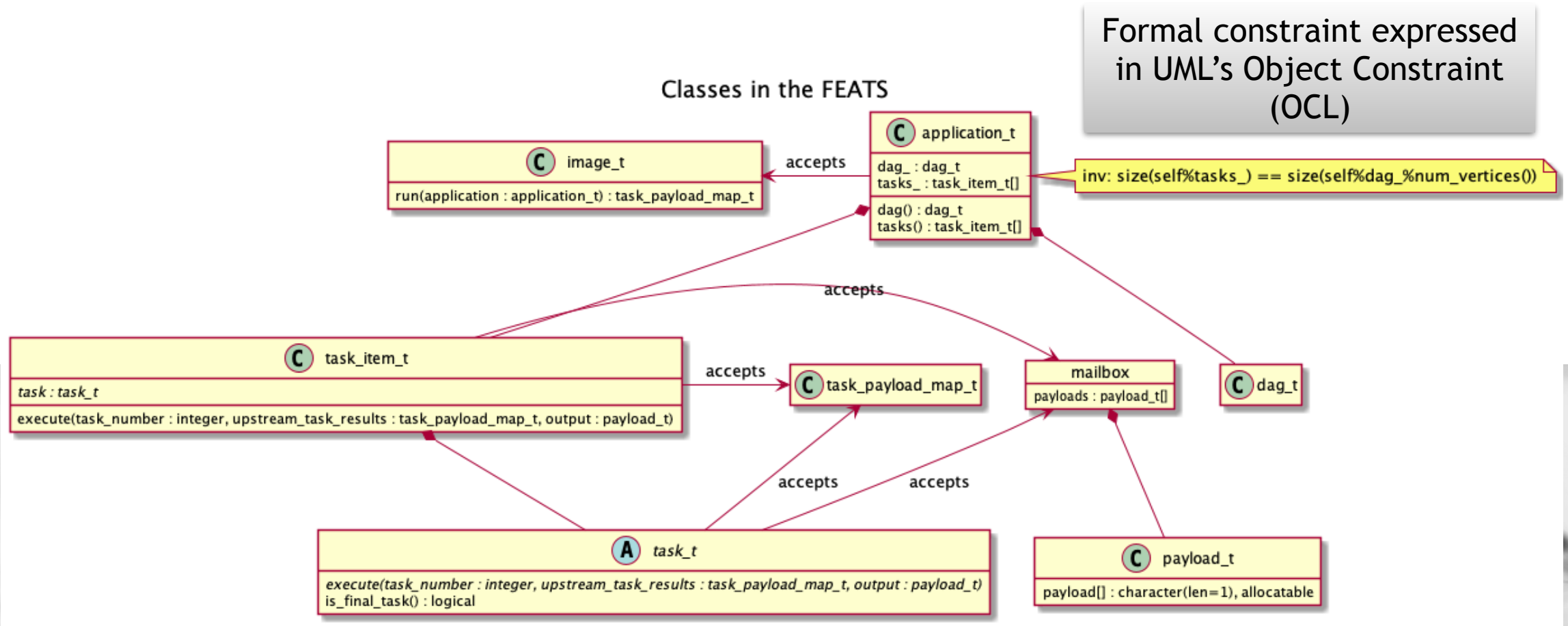
FEATS Parallel Traversal of the FEATS DAG



A terminal window titled "feats -- -zsh -- 121x34" displays the output of a command. The prompt is "[rouson:~/Repositories/sourceryinstitute/feats]". The output shows "main+*" in green, followed by "39s" in green, "130" in purple, and a plus-minus symbol in blue. The rest of the terminal is empty.

```
[rouson:~/Repositories/sourceryinstitute/feats] main+* 39s 130 ±
```

UML Class Diagram with a Simple Contract



Application_t Interface & Implementation

```
application_m.f90
9  type application_t
10  !! A complete representation of an application that can be executed
11  private
12  type(dag_t) dag_ !! Describes the dependencies between tasks
13  type(task_item_t), allocatable :: tasks_(:) !! tasks to be executed
14  contains
15  private
16  procedure, public :: dag
17  procedure, public :: tasks
18  end type
19
20  • interface application_t
21
22  pure module function construct(dag, tasks) result(application)
23  implicit none
24  type(dag_t), intent(in) :: dag
25  type(task_item_t), intent(in) :: tasks(:)
26  type(application_t) application
27  end function
28
29  end interface
30
31  • interface
32
33  pure module function dag(self)
34  implicit none
35  • class(application_t), intent(in) :: self
36  • type(dag_t) :: dag
37  end function
38
39  pure module function tasks(self)
40  implicit none
41  • class(application_t), intent(in) :: self
42  • type(task_item_t), allocatable :: tasks_(:)
43  • end function
44
45  end interface
46
47  end module application_m

application_s.f90
1  submodule(application_m) application_s
2  use assert_m, only : assert
3  implicit none
4
5  contains
6
7  module procedure construct
8  call assert(size(tasks)==dag%num_vertices(), &
9  |'application(construct): size(tasks)==dag%num_vertices()')
10  application%dag_ = dag
11  application%tasks_ = tasks
12  end procedure
13
14  module procedure dag
15  dag = self%dag_
16  end procedure
17
18  module procedure tasks
19  tasks = self%tasks_
20  end procedure
21
22  end submodule application_s
```

fpm build src/application_s.f90* 11 0 0 9:7 LF UTF-8 Fortran - Free Form main Fetch GitHub Git (2)

Application Construction


```
application_generator_m.f90
73      block
74          character(len=*),      parameter :: external_ = 'shape=square,fillcolor="green",style=filled'
75          character(len=*),      parameter :: root      = 'shape=circle,fillcolor="white",style=filled'
76          character(len=*),      parameter :: branch    = 'shape=square,fillcolor="SlateGray1",style=filled'
77          character(len=len(branch)), parameter :: leaf  = 'shape=circle,fillcolor="cornsilk",style=filled'
78          type(dag_t) feats
79          integer i
80          type(varying_string) name_string(size(names))
81          type(task_item_t), allocatable :: tasks(:)
82
83          name_string = var_str(names)
84
85          feats = &
86              dag_t([ &
87                  vertex_t([integer::],      name_string(assert_m),      var_str(external_)) &
88                  , vertex_t([integer::],      name_string(dag_m),        var_str(external_)) &
89                  , vertex_t([integer::],      name_string(payload_m),    var_str(leaf)      ) &
90                  , vertex_t([integer::],      name_string(compile_m),    var_str(leaf)      ) &
91                  , vertex_t([integer::],      name_string(data_loc_map_m), var_str(leaf)      ) &
92                  , vertex_t([payload_m],      name_string(task_m),        var_str(branch)    ) &
93                  , vertex_t([task_m],         name_string(task_item_m),  var_str(leaf)      ) &
94                  , vertex_t([dag_m, task_item_m], name_string(app_m),          var_str(branch)    ) &
95                  , vertex_t([app_m, dag_m, task_item_m, compile_m], name_string(app_generator_m), var_str(branch)    ) &
96                  , vertex_t([app_m, data_loc_map_m], name_string(image_m),      var_str(branch)    ) &
97                  , vertex_t([app_generator_m, image_m], name_string(main),         var_str(root)       ) &
98                  , vertex_t([task_item_m],      name_string(task_item_s),  var_str(root)       ) &
99                  , vertex_t([compile_m],        name_string(compile_s),   var_str(branch)    ) &
100                 , vertex_t([app_generator_m],   name_string(app_generator_s), var_str(root)      ) &
101                 , vertex_t([data_loc_map_m],    name_string(data_loc_map_s), var_str(root)      ) &
102                 , vertex_t([payload_m],         name_string(payload_s),   var_str(root)       ) &
103                 , vertex_t([app_m, assert_m],   name_string(app_s),       var_str(root)       ) &
104                 , vertex_t([payload_m],         name_string(mailbox_m),   var_str(branch)    ) &
105                 , vertex_t([image_m, mailbox_m], name_string(image_s),     var_str(root)       ) &
106                 , vertex_t([data_loc_map_m, payload_m, task_m], name_string(final_task_m), var_str(branch)    ) &
107                 , vertex_t([final_task_m],      name_string(final_task_s), var_str(root)       ) &
108             ])
109          tasks = [(task_item_t(compile_task_t(name_string(i))), i = 1, size(names))]
110          application = application_t(feats, tasks)
111      end block
---
```

1 fm build example/application_generator_m.f90 1 1

• LF UTF-8 Fortran - Free Form main Fetch GitHub Git (5)

A simple yet slightly non-obvious assertion utility for Fortran 2018

- 🧙 To promote contract enforcement via runtime constraint checking.
- 🧙 To mitigate a commonly cited reason for not writing **pure** procedures: difficulty production output.
- 🧙 To facilitate toggling assertions on or off for debugging or production builds, respectively.



[sourceryinstitute / assert](#)
Public

Unwatch 2
Star 10
Fork 3

[Code](#)
[Issues](#)
[Pull requests 1](#)
[Actions](#)
[Projects](#)
[Wiki](#)
[Security](#)
[Insights](#)

main
2 branches
4 tags

[Go to file](#)
[Add file](#)
[Code](#)


rouson Update README.md
24a667b yesterday
46 commits

.github/workflows	fix(ci): accidental extra line	2 days ago
doc	feat: add PlantUML diagram for classes in Assert	29 days ago
example	refac: reorganize/rename files/directories	last month
src	feat: add multidensional intrinsic_array support	15 days ago
test/unit-tests	fix(src,test): workaround no NAG co_reduce support	15 days ago
.gitignore	doc(ford): store/gitignore files in doc/html	15 days ago
LICENSE	Initial commit	last month
README.md	Update README.md	yesterday
doc-generator.md	doc(ford): store/gitignore files in doc/html	15 days ago
fpm.toml	feat: assertion utility, example, & fpm build file	last month

README.md

Assert

A simple assertion utility taking advantage of the Fortran 2018 standard's introduction of variable stop codes and error termination inside pure procedures.

Motivations

- To mitigate against a reason developers often cite for not writing `pure` procedures: their inability to produce output in normal execution.
- To promote the enforcement of programming contracts.

About

A simple yet slightly non-obvious assertion utility for Fortran 2018

[Readme](#)

[BSD-3-Clause License](#)

Releases 4

[Build with the NA...](#) Latest
10 days ago


[+ 3 releases](#)


Packages

No packages published

[Publish your first package](#)

Contributors 2


rouson Damian Rouson


everythingfunctional Br...

Environments 1

[github-pages](#) Active

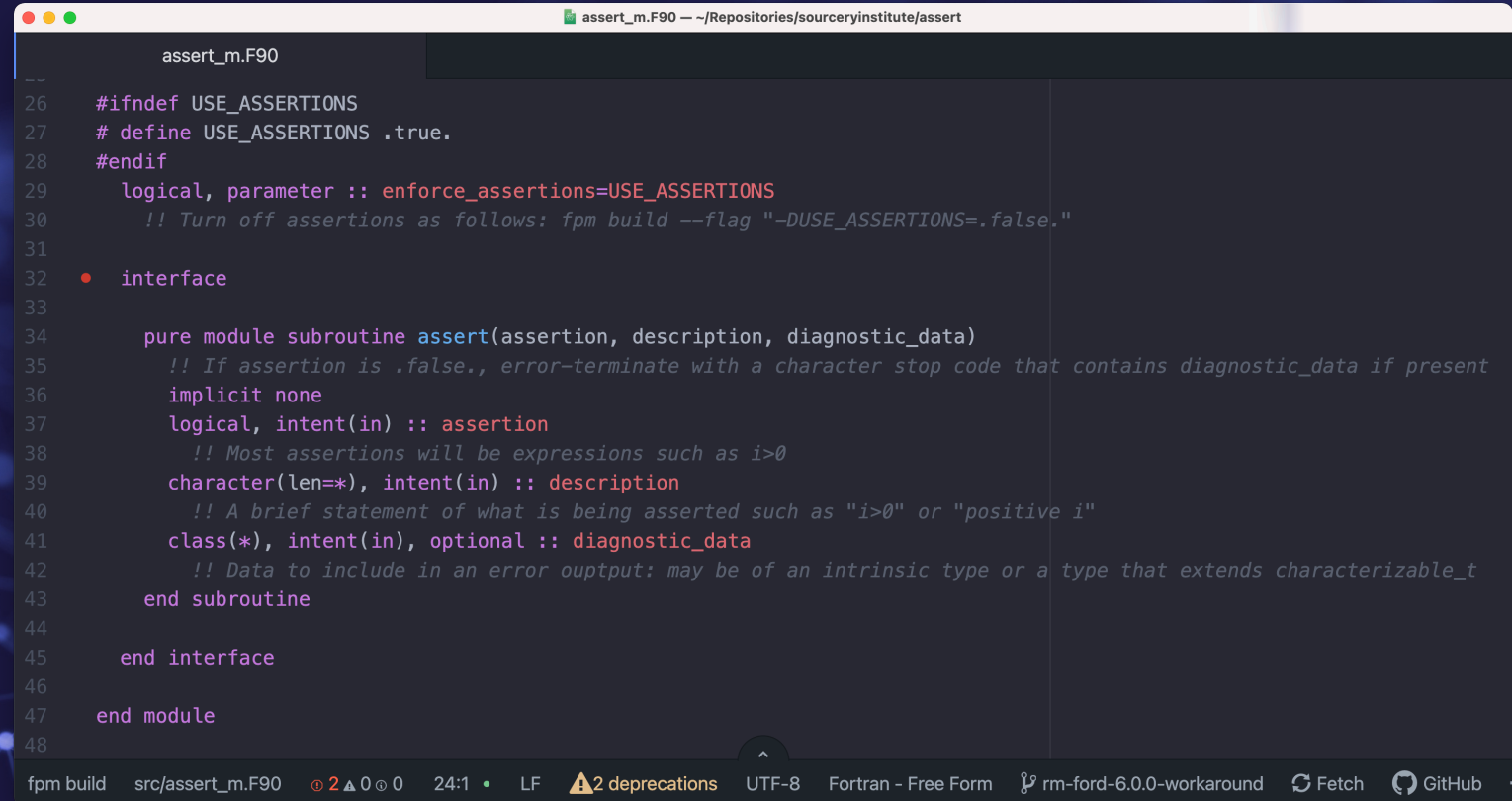
Languages

Assert

A simple yet slightly non-obvious assertion utility for Fortran 2018

Approach:

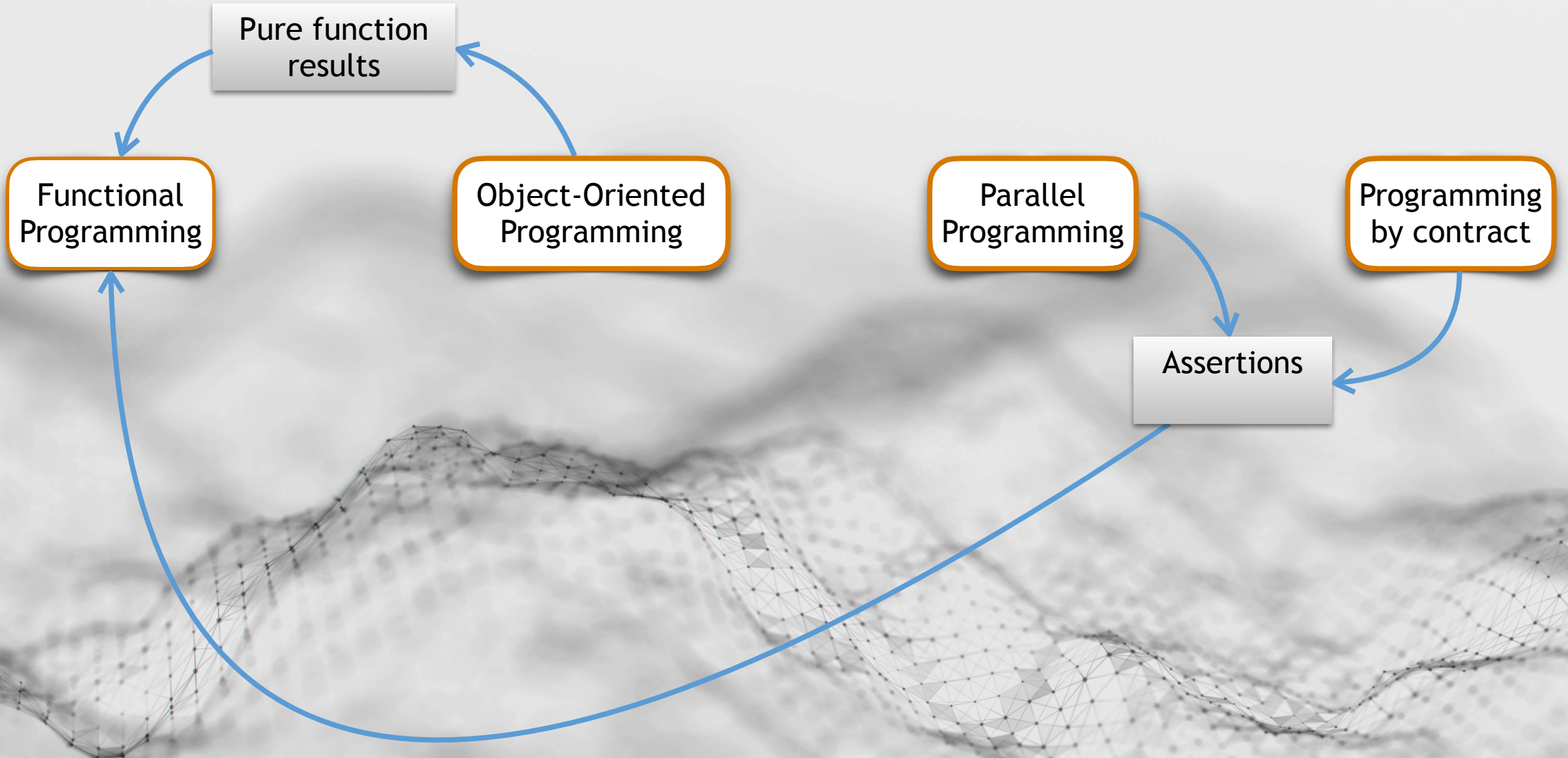
- Express constraints in logical expression as actual arguments to a pure assert subroutine.
- Execute **error stop** upon assertion failure.
- Bracket all executable statements in in the `assert()` subroutine by a conditional evaluation of a parameter logical `ASSERTIONS` variable
→ facilitate dead-code removal optimization



```
assert_m.F90
--
26 #ifndef USE_ASSERTIONS
27 # define USE_ASSERTIONS .true.
28 #endif
29 logical, parameter :: enforce_assertions=USE_ASSERTIONS
30 !! Turn off assertions as follows: fpm build --flag "-DUSE_ASSERTIONS=.false."
31
32 • interface
33
34 pure module subroutine assert(assertion, description, diagnostic_data)
35 !! If assertion is .false., error-terminate with a character stop code that contains diagnostic_data if present
36 implicit none
37 logical, intent(in) :: assertion
38 !! Most assertions will be expressions such as i>0
39 character(len=*), intent(in) :: description
40 !! A brief statement of what is being asserted such as "i>0" or "positive i"
41 class(*), intent(in), optional :: diagnostic_data
42 !! Data to include in an error ouptput: may be of an intrinsic type or a type that extends characterizable_t
43 end subroutine
44
45 end interface
46
47 end module
48
```

fpm build src/assert_m.F90 2 0 0 24:1 • LF 2 deprecations UTF-8 Fortran - Free Form rm-ford-6.0.0-workaround Fetch GitHub

Synergies at the Intersection



Assert

A simple yet slightly non-obvious assertion utility for Fortran 2018

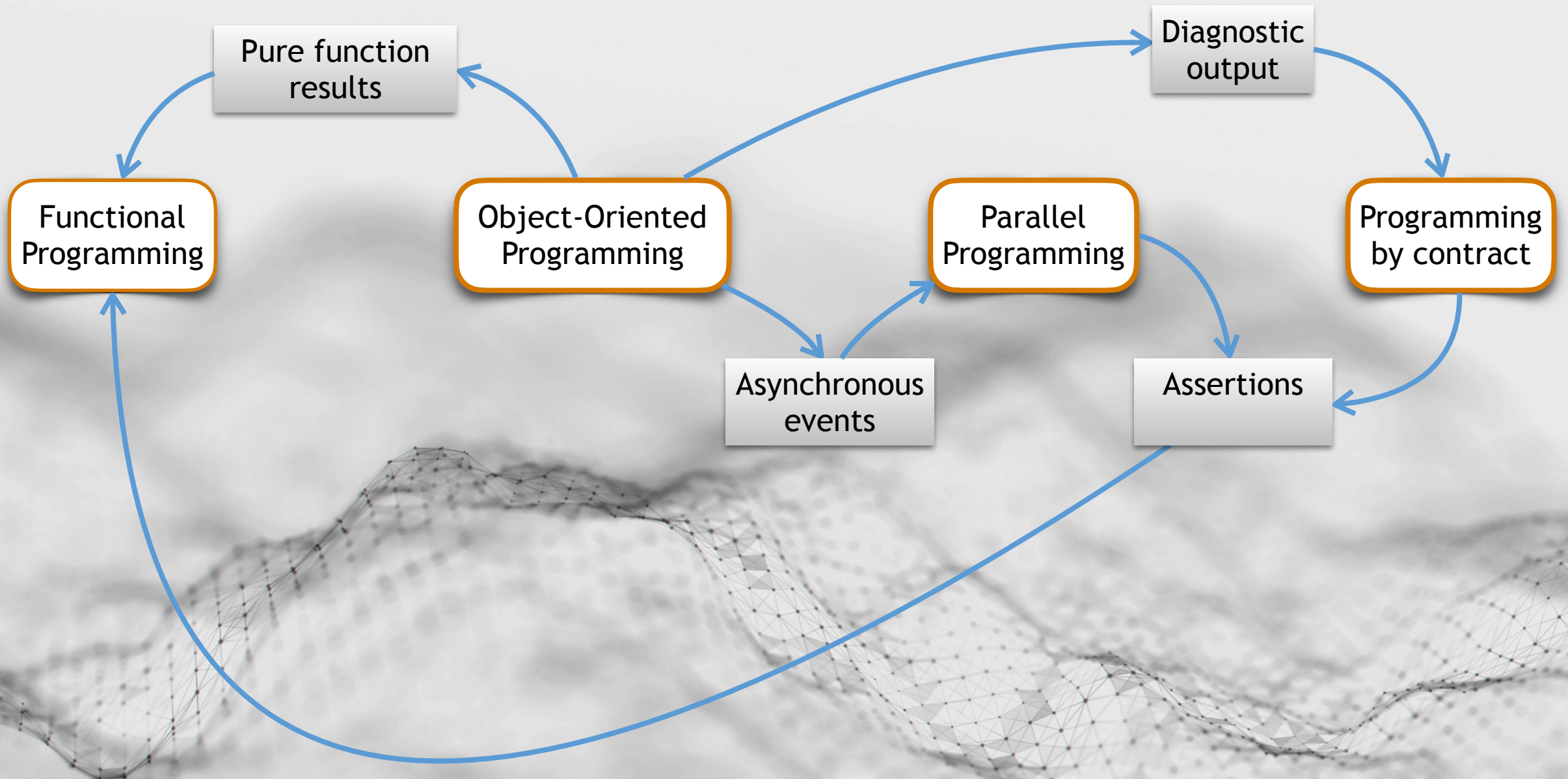
Approach:

- 🦋 Express constraints in logical expression as actual arguments to a pure assert subroutine.
- 🦋 Execute **error stop** upon assertion failure.
- 🦋 Bracket all executable statements in in the assert() subroutine by a conditional evaluation of a parameter logical ASSERTIONS variable
 - facilitate dead-code removal optimization
- 🦋 Optionally output diagnostic data as scalar default character expression stop codes.

```
assert_s.f90
12 module procedure assert
13 • use characterizable_m, only : characterizable_t
14
15 character(len=:), allocatable :: header, trailer
16
17 toggle_assertions: &
18 if (enforce_assertions) then
19
20 check_assertion: &
21 if (.not. assertion) then
22
23 associate(me=>this_image()) ! work around gfortran bug
24 header = 'Assertion "' // description // '" failed on image ' // string(me)
25 end associate
26
27 represent_diagnostics_as_string: &
28 if (.not. present(diagnostic_data)) then
29
30 trailer = "(none provided)"
31
32 else
33
34 select type(diagnostic_data)
35 type is(character(len=:))
36 trailer = diagnostic_data
37 type is(complex)
38 trailer = string(diagnostic_data)
39 type is(integer)
40 trailer = string(diagnostic_data)
41 type is(logical)
42 trailer = string(diagnostic_data)
43 type is(real)
44 trailer = string(diagnostic_data)
45 class is(characterizable_t)
46 trailer = diagnostic_data%as_character()
```

fpm build src/assert_s.f90 0 3 ▲ 0 0 0 1:1 • LF ⚠ 2 deprecations UTF-8 Fortran - Free Form 📄 rm-ford-6.0.0-workaround 🔄 Fetch 🌐 GitHub

Synergies at the Intersection



Abstract Calculus Pattern

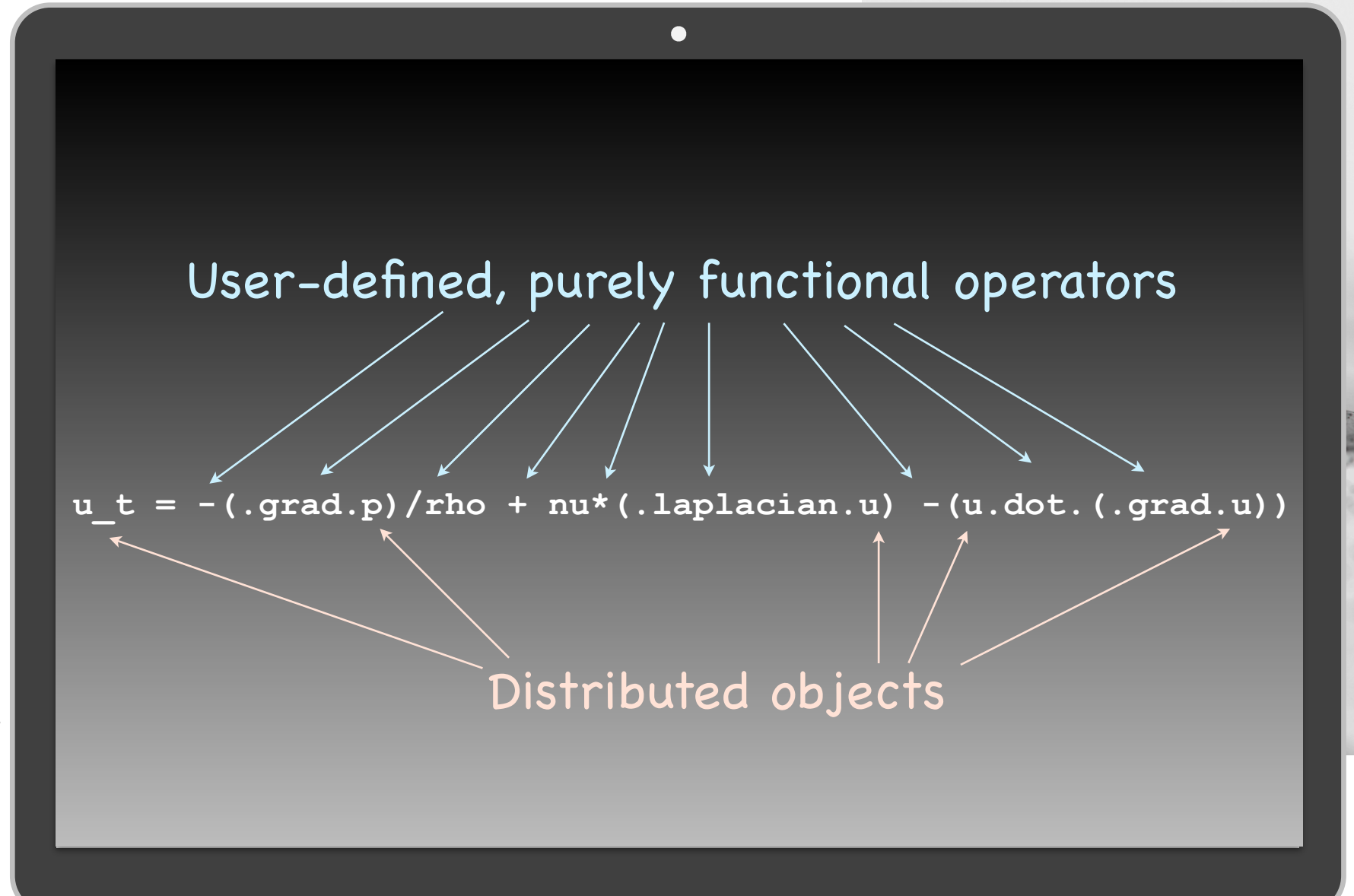
Standard semantic requirement:
operands must have the
`intent(in)` attribute.

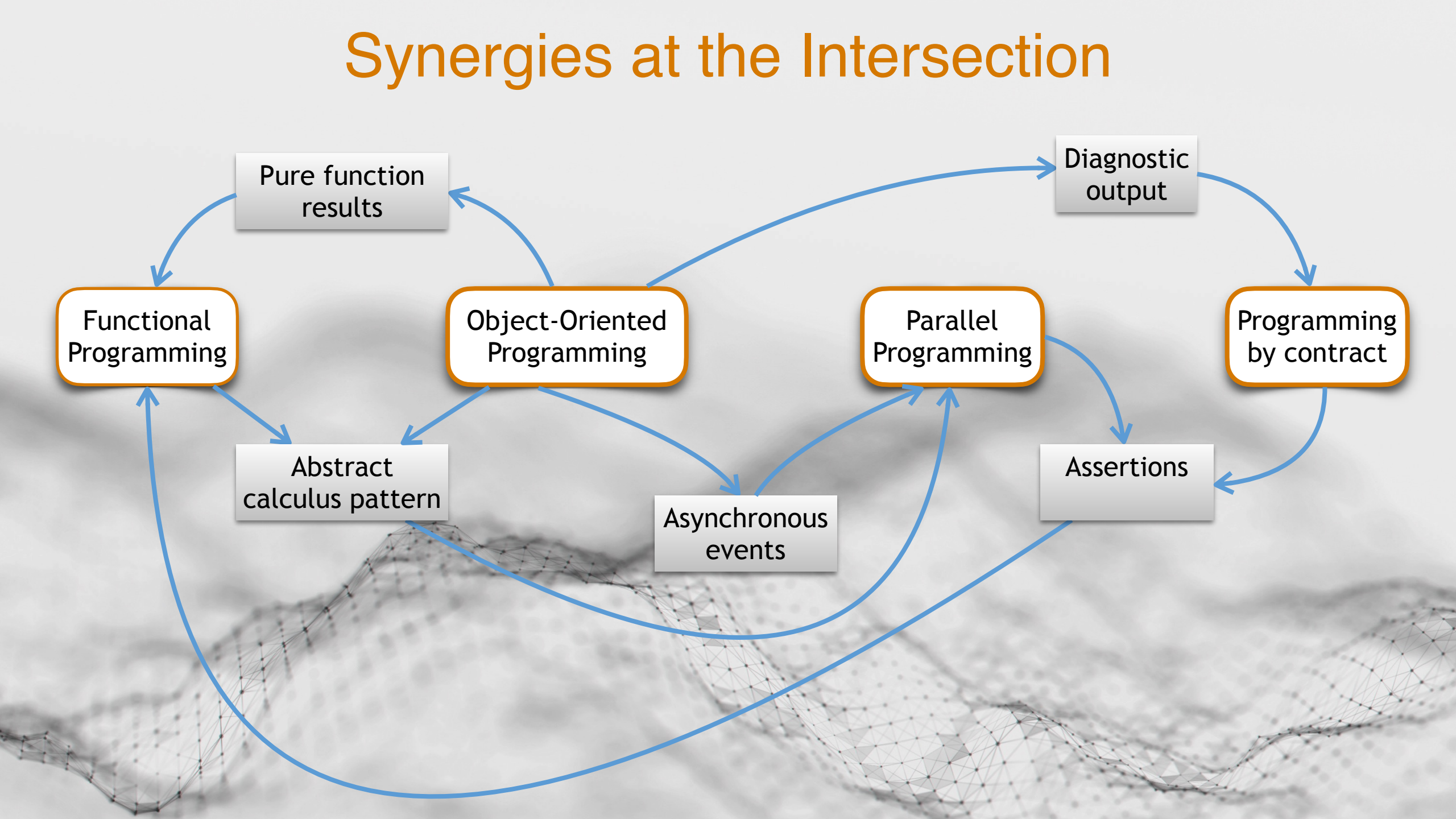
→ not merely syntactic sugar.

A nudge in the direction of
functional programming?

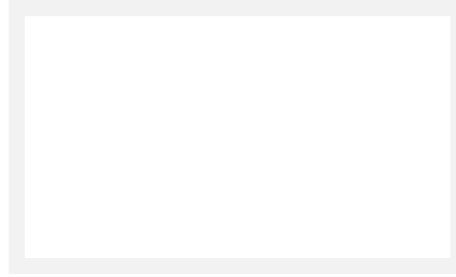
Performance implication: fully
asynchronous expression evaluation
even with communication.

Image control happens in the one
non-functional step: the assignment.



[illegible]

Outline



● Introduction

● Case Studies

● Intersectionality: Notes from an underdog

- ✻ A social science definition.
- ✻ A diverse workforce does work differently...
- ✻ And does different.

● Acknowledgments



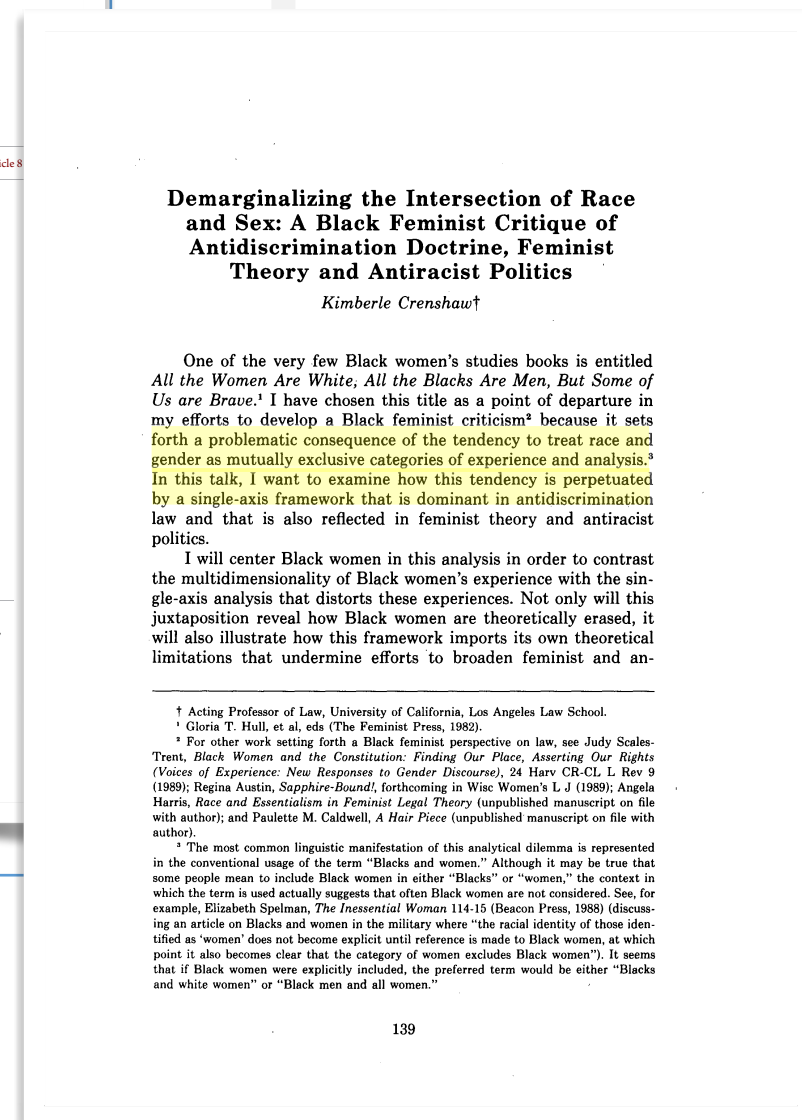


Prof. Kimberle Crenshaw
UCLA School of Law

Intersectionality



Google Scholar citation
count: 21,620



An Underdog

University of Virginia, Department of Computer Science
CS655: Programming Languages, Spring 2001

How do we tell truths that might hurt?

Edsger W.Dijkstra, 18 June 1975

from <https://www.cs.utexas.edu/users/EWD/ewd04xx/EWD498.PDF>

Sometimes we discover unpleasant truths. Whenever we do so, we are in difficulties: suppressing them is scientifically dishonest, so we must tell them but telling them, however, will fire back on us. If the truths are sufficiently palatable, our audience is psychically incapable of accepting them and we will be written off as totally unrealistic, hopelessly idealistic, dangerously revolutionary, foolishly gullible or what have you. (Besides that, telling such truths is a sure way of making oneself unpopular in many circles, and, as such, it is an act that, in general, is not without personal risks. Vide Galileo Galilei.....)

Computing Science seems to suffer severely from this conflict. On the whole it remains silent and tries to escape this conflict by shifting its attention. (For instance: with respect to COBOL you can really do only one of two things: fix the disease or pretend that it does not exist. Most Computer Science Departments have opted for the latter easy way out.) But, Brethern, I ask you: is this honest? Is not our prolonged silence fretting away Computing Science's intellectual integrity? Are we decent by remaining silent? If not, how do we speak up?

To give you some idea of the scope of the problem I have listed a number of such truths. (Nearly all computing scientists I know well will agree without hesitation to nearly all of them. Yet we allow the world to behave as if we do not know them....)

- Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.
- The easiest machine applications are the technical/scientific computations.
- The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.
- FORTRAN --"the infantile disorder"--, by now nearly 20 years old, is hopelessly inadequate for whatever computer application you have in mind today: it is now too clumsy, too risky, and too expensive to use.

1975-2001



CODING HORROR
programming and human factors

ENHANCED BY Google

22 Apr 2005

You Can Write FORTRAN in any Language

A recent [user-submitted CodeProject article](#) took an interesting perspective on the [VB.NET/C# divide](#) by proposing that the **culture of Visual Basic** is not conducive to professional software development:

We've seen that the cultures of VB and C# are very different. And we've seen that this is no fault of the programmers that use them. Rather this is a product of the combination of factors that collectively could be called their upbringing -- business environment, target market, integrity and background of the original language developers, and a myriad other factors.

2005



COVID-19
Jobs &
Resources

Search
Jobs

News

Advice

Register

Sign
In

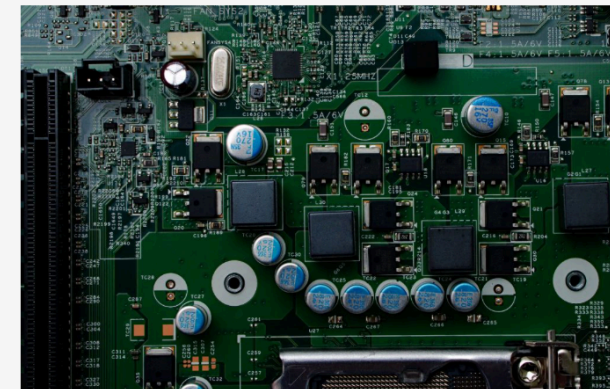
Recruiters

Select Region



Real programmers write Java like FORTRAN

by Frank Becker 05 January 2021



I'm a Java coder and I work on high speed trading systems. I know that people dispute whether Java is the best language for really low latency work, but my experience is that it is absolutely viable. I have seen optimized Java code that is only about 20-30% slower than very, very optimized C code and this is pretty awesome. The application was able to respond to market signals within single digit microseconds every time.

Real programmers can write FORTRAN in any language. The issue with Java isn't that you can't write low latency code. It's that you are left almost completely without tools. There's very little you can use from Java standard libraries. You're therefore left scratching your head about how to solve even the simplest problem like managing memory without something coming and suddenly creating a lot of latency where you don't want it to happen.

If you can code a FORTRAN-like Java, you can overcome this. And Java has multiple advantages, as follows:

1. Java is a simpler language than C++, but it allows for easy object modelling, which is not available in plain C. The smaller feature set of the language also helps the developers stay focused on the logic of the application rather than on expressing their technical superiority through application of all the bells and whistles available in the standard (which often makes C++ code hard to read and improve by others).

2021

"Music lovers are at high risk of being inspired by this exploration of the connections between music and physics." —*Wall Street Journal*

$E = -j \sum S_i S_{i+1}$

$v^2 = \frac{g}{\rho}$

THE SECRET LINK BETWEEN
MUSIC AND THE STRUCTURE
OF THE UNIVERSE

$F = \text{constant} = m \frac{d^2}{dt^2} X(t)$

$\lambda = \frac{nh}{p}$

$E = hf$

$\frac{\partial^2 u(x,t)}{\partial t^2} = v^2 \frac{\partial^2 u(x,t)}{\partial x^2}$

$E = \sum_{ij} w_{ij} n_i n_j$

$\frac{\partial^2 X(t)}{\partial t^2} = \alpha X(t)$

THE SECRET LINK BETWEEN
MUSIC AND THE STRUCTURE
OF THE UNIVERSE

$\lambda = v/f$

$i\hbar \frac{\partial \Psi(x,t)}{\partial t} = \mathcal{H} \Psi(x,t)$

$F = \text{constant} = m \frac{d^2}{dt^2} X(t)$

STEPHON
ALEXANDER

$f(t) = \sum_{n=0}^{\infty} A_n \sin(\omega_n t) = A_0 \sin(\omega_0 t) + A_1 \sin(\omega_1 t) + \dots$

$\lambda = v/f$

What a great compliment, an affirmation and acknowledgment of the improvisational, inclusive, cultural, and intellectual contributions of this music called jazz.”

“It occurred to me by intuition, and music was the driving force behind that intuition. My discovery was the result of musical perception.”

- Albert Einstein (when asked about his theory of relativity)

... and does different work

<https://collegeville.github.io/CW20/>

Collegeville 2020 Home | ["Collegeville Workshop 2020"]

Better Scientific Software


Collegeville Workshop 2020

CW20 brings community members together to advance developer productivity for scientific software

[View On GitHub](#)

This project is maintained by [Collegeville](#)

Collegeville 2020 Home



2020 Collegeville Workshop
on Scientific Software
Developer Productivity
July 21 - 23, 2020

Details

- [Workshop Survey](#)
- [Agenda Overview](#)
 - [Tuesday, July 21 Detailed Agenda](#)
 - [Wednesday, July 22 Detailed Agenda](#)
 - [Thursday, July 23 Detailed Agenda](#)
- [Resources](#)
 - [Posters](#)
 - [Recordings](#)

[bssw.io/blog_posts/increasing-productivity-](#)

Collegeville 2020 Home | ["Collegeville Workshop 2020"]

Better Scientific Software


[Information For](#) [Contribute to BSSw](#) [Receive Our Email Digest](#) [Contact BSSw](#)

[Resources](#) [Blog](#) [Events](#) [About](#)

[HOME](#) > [BLOG](#) >

Increasing Productivity by Broadening Participation in Scientific Software Communities

SHARE in f t



ALEXANDRA BALLOW, A STUDENT AT YOUNGSTOWN STATE UNIVERSITY WHO WAS A PARTICIPANT IN THE BROADER ENGAGEMENT PROGRAM AT SIAM CSE19, PRESENTING HER WORK TO PAUL HOVLAND OF ARGONNE NATIONAL LAB. ALEXANDRA PREVIOUSLY PARTICIPATED IN THE SUSTAINABLE RESEARCH PATHWAYS PROGRAM.

PUBLISHED SEP 25, 2020

AUTHOR [MARY ANN LEUNG](#), [DAMIAN ROUSON](#), AND [LOIS CURFMAN MCINNES](#)

TOPICS [BETTER COLLABORATION](#) [STRATEGIES FOR MORE EFFECTIVE TEAMS](#) [FUNDING SOURCES AND PROGRAMS](#)

[bssw.io/blog_posts/increasing-productivity-](#)

Collegeville 2020 Home | ["Collegeville Workshop 2020"]

Better Scientific Software

Numerous studies have shown that diverse organizations, teams, and communities perform more creatively and effectively—and thus are more productive. While some efforts are already under way to broaden participation in high-performance computing (HPC) and computational science and engineering (CSE), we observe that our communities could benefit by increasing emphasis on sustainable strategies to advance diversity and inclusion.

Lois Curfman McInnes: Mary Ann and Damian, many thanks for providing your perspectives as leaders who are working to broaden participation of under-represented groups in high-performance scientific computing. I am hoping to learn more about how individuals and groups can take steps forward within our own spheres of influence to broaden participation and thereby to help the community as a whole advance in productivity.

Lois: What are your backgrounds?

Damian Rouson: My training is in computational fluid dynamics. As a consultant, educator, and researcher, I have long aimed to adapt leading-edge software engineering practices to computational science and engineering applications. I'm passionate about advancing development practices in modern Fortran. As an African-American, I see my adopting an underdog language as partly an extension of being outside the dominant culture in STEM fields. At a time 20 years ago when most people who were passionate about improving scientific software development were adopting other languages, I found that an ability to embrace difference and combat stigma along one obvious dimension, ethnicity, makes it feel natural to swim upstream or outside the mainstream along another dimension: programming language choice.

Mary Ann Leung: My training is in computational quantum mechanics simulations on HPC systems. As a woman of color, a first generation scientist, and a non-traditional student, I found the need for and became interested in diversity in science during school. I got involved in diversity initiatives and founded a few campus organizations focused on diversity and inclusion as well as career and professional development. I later ended up migrating my career to workforce development where my passion for the people side of science could be realized.

Lois: Why is broadening participation important for improving productivity – of software developers and high-performance computational science overall?

Mary Ann: CSE/HPC developer productivity can be advanced by engaging a broader set of individuals for several important reasons. First off, CSE and HPC are inherently complex and require teamwork, creative solutions, and collaboration. Research indicates that diverse teams are more innovative. Additionally, the workforce in general is becoming more diverse, and by not including members of underrepresented groups, we are missing out on potential new developers with new ideas and approaches.

Lois: What are some issues that organizations should consider in order to create

Acknowledgements

- 🐦 Zaak Beekman, ParaTools, Inc.
- 🐦 Janne Blomquist, GCC
- 🐦 Aurelien Bouteiller, University of Tennessee
- 🐦 Tobias Burnus, Germany
- 🐦 C. Carson, Stanford University
- 🐦 Alessandro Fanfarillo, AMD
- 🐦 Salvatore Filippone, University of Rome Tor Vergata
- 🐦 Brian Friesen, Berkeley Lab
- 🐦 Daniel Celis Garza, Oxford University
- 🐦 Ethan Gutmann, National Center for Atmospheric Research
- 🐦 Magne Haveraaen, U. Bergen
- 🐦 Jeff Hammond, NVIDIA
- 🐦 LaHaine (OpenCoarrays contributor)
- 🐦 Briana Lang, Working Lunch, LLC
- 🐦 Brian Laubacher, Autoliv
- 🐦 Elias Lettl, University of Augsburg
- 🐦 Karla Morris, Sandia National Laboratories, Archaeologic, and Sourcery Institute
- 🐦 Hari Radhakrishnan, NIPD Genetics
- 🐦 Harris Snyder, Archaeologic Inc.
- 🐦 Katherine Rasmussen, Archaeologic Inc. and Sourcery Institute
- 🐦 Patrick Raynaud, U.S. Nuclear Regulatory Commission
- 🐦 Brad Richardson, Archaeologic, Sourcery Institute, & Everything Functional
- 🐦 Soren Rasmussen, Cranfield University and National Center for Atmospheric Research
- 🐦 Anuj Sharma (OpenCoarrays contributor)
- 🐦 Sameer Shende, U. Oregon and ParaTools, Inc.
- 🐦 Robert Singleterry, NASA Langley Research Center
- 🐦 Andre Verheschild, Badger Systems
- 🐦 Nathan Weeks, Iowa State University
- 🐦 Jacob Williams, NASA Johnson Space Flight Center