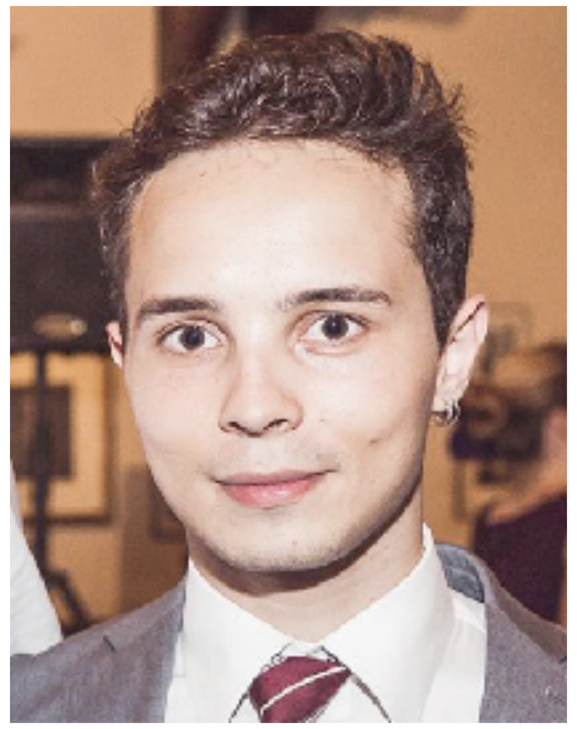
Lightweight code verification for science

Dominic Orchard

Cross-VESRI Journal Club - 13th June 2023

UNIVERSITY OF CAMBRIDGE
Institute of Computing for Climate Science
University of Kent
Programming Languages and Systems for Science laboratory

Thanks to the work of:



Mistral Contrastin

Matthew Danish

Andrew Rice

Ben Orchard

And thanks to the support of:

Bloomberg

UKRI Engineering and Physical Sciences Research Council

SCHMIDT FUTURES

Procedia Computer Science

Volume 29, 2014, Pages 713–727

ICCS 2014. 14th International Conference on Computational Science

A computational science agenda for programming language research

Dominic Orchard[1], Andrew Rice[2]

[1] Computer Laboratory, University of Cambridge
dominic.orchard@cl.cam.ac.uk
[2] Computer Laboratory, University of Cambridge
andrew.rice@cl.cam.ac.uk

Abstract

Scientific models are often expressed as large and complicated programs. These programs embody numerous assumptions made by the developer (*e.g.*, for differential equations, the discretization strategy and resolution). The complexity and pervasiveness of these assumptions means that often the only true description of the model is the software itself. This has led various researchers to call for scientists to publish their source code along with their papers. We argue that this is unlikely to be beneficial since it is almost impossible to separate implementation assumptions from the original scientific intent. Instead we advocate higher-level abstractions in programming languages, coupled with lightweight verification techniques such as specification and type systems. In this position paper, we suggest several novel techniques and outline an evolutionary approach to applying these to existing and future models. One-dimensional heat flow is used as an example throughout.

Keywords: computational science, modelling, programming, verification, reproducibility, abstractions, type systems, language design

Journal of Computational Science 9 (2015) 156–162

Contents lists available at ScienceDirect

Journal of Computational Science

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

Evolving Fortran types with inferred units-of-measure

Dominic Orchard[a,*], Andrew Rice[b], Oleg Oshmyan[b]

[a] Department of Computing, Imperial College London, United Kingdom
[b] Computer Laboratory, University of Cambridge, United Kingdom

A R T I C L E   I N F O

Article history:
Available online 18 April 2015

Keywords:
Units-of-measure
Dimension typing
Type systems
Verification
Code base evolution
Fortran
Language design

A B S T R A C T

Dimensional analysis is a well known technique for checking the consiste
physical quantities, constituting a kind of type system. Various type system
and its refinement to units-of-measure, have been proposed. In this pape
implementation of a units-of-measure system for Fortran, provided as a pr
designed to aid adding units to existing code base: units may be polymorph
thermore, we introduce a technique for reporting to the user a set of *critica*
explicitly annotated with units to get the maximum amount of unit informat
ber of explicit declarations. This aids adoption of our type system to existing
are many in computational science projects.

1. Introduction

Type systems are one of the most popular static techniques for recognizing and rejecting large classes of programming error. A common analogy for types is of physical quantities (*e.g.*, in [2]), where type checking excludes, for example, the non-sensical addition of non-comparable quantities such as adding 3 m to 2 J; they have different *dimensions* (length *vs.* energy) and different *units* (metres *vs.* joules). This analogy between types and dimensions/units goes deeper. The approach of *dimensional analysis* checks the consistency of formulae involving physical quantities, acting as a kind of type system (performed by hand, long before computers). Various automatic type-system-like approaches have been proposed for including dimensional analysis in programming languages (*e.g.* [10] is a famous paper detailing one such approach,

circumstances. It therefore seems inevita
likely in computational science too.

The importance of units is often directly
code. We have seen source files careful
units and dimensions of each variable and
watched programmers trying to use this i
scrolling up and down, repeatedly referri
tion of each parameter. Incorporating un
would move the onus of responsibility f
the compiler.

A recent ISO standards proposal (N196
a units-of-measure system which follov
explicitness [7]. Every variable declaratio
unit declaration and every composite
seconds) must itself be explicitly declared

Procedia Computer Science
Volume 29, 2014, Pages 713–727
ICCS 2014. 14th International Conference on Computational Science

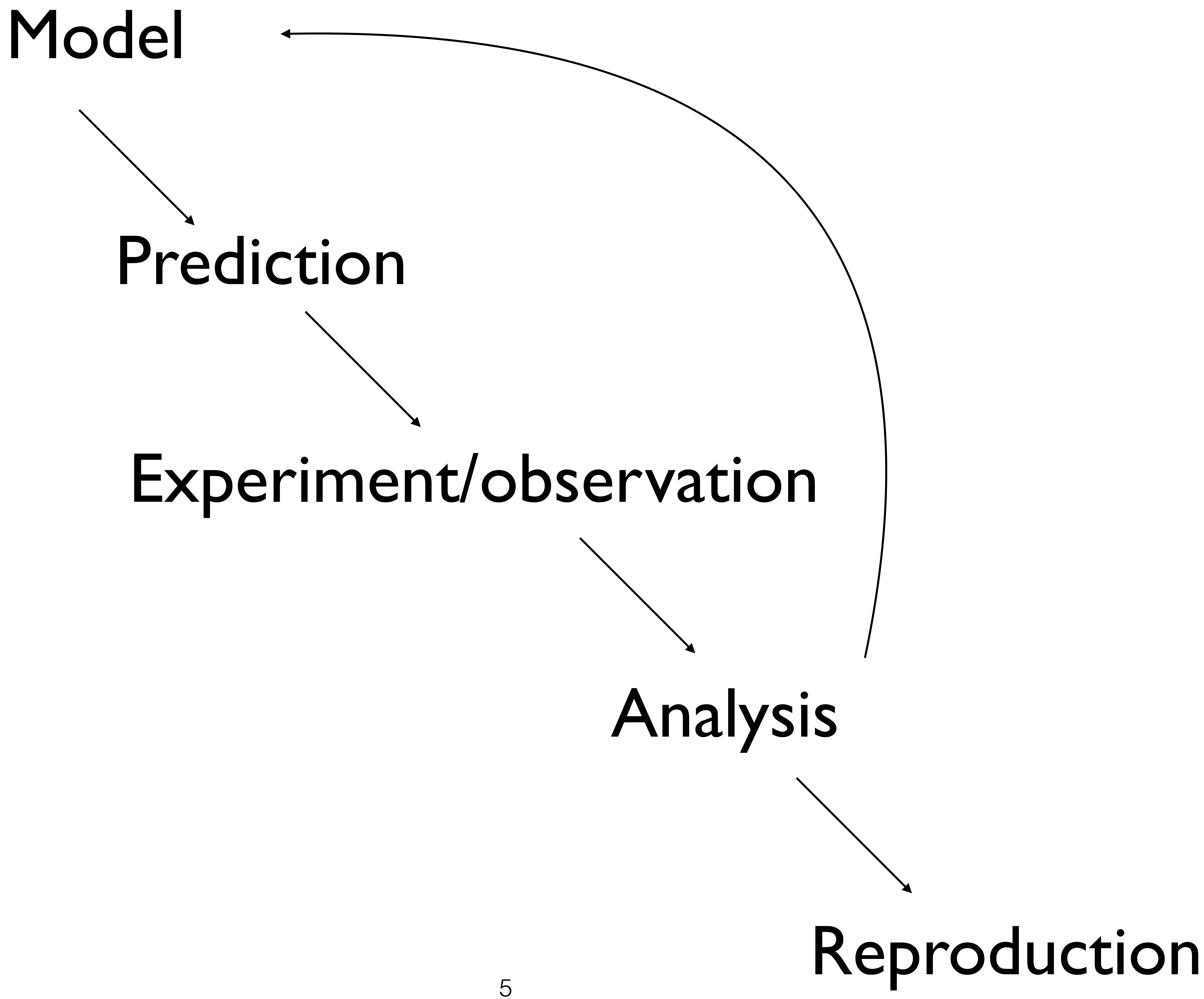
A computational science agenda for programming language research

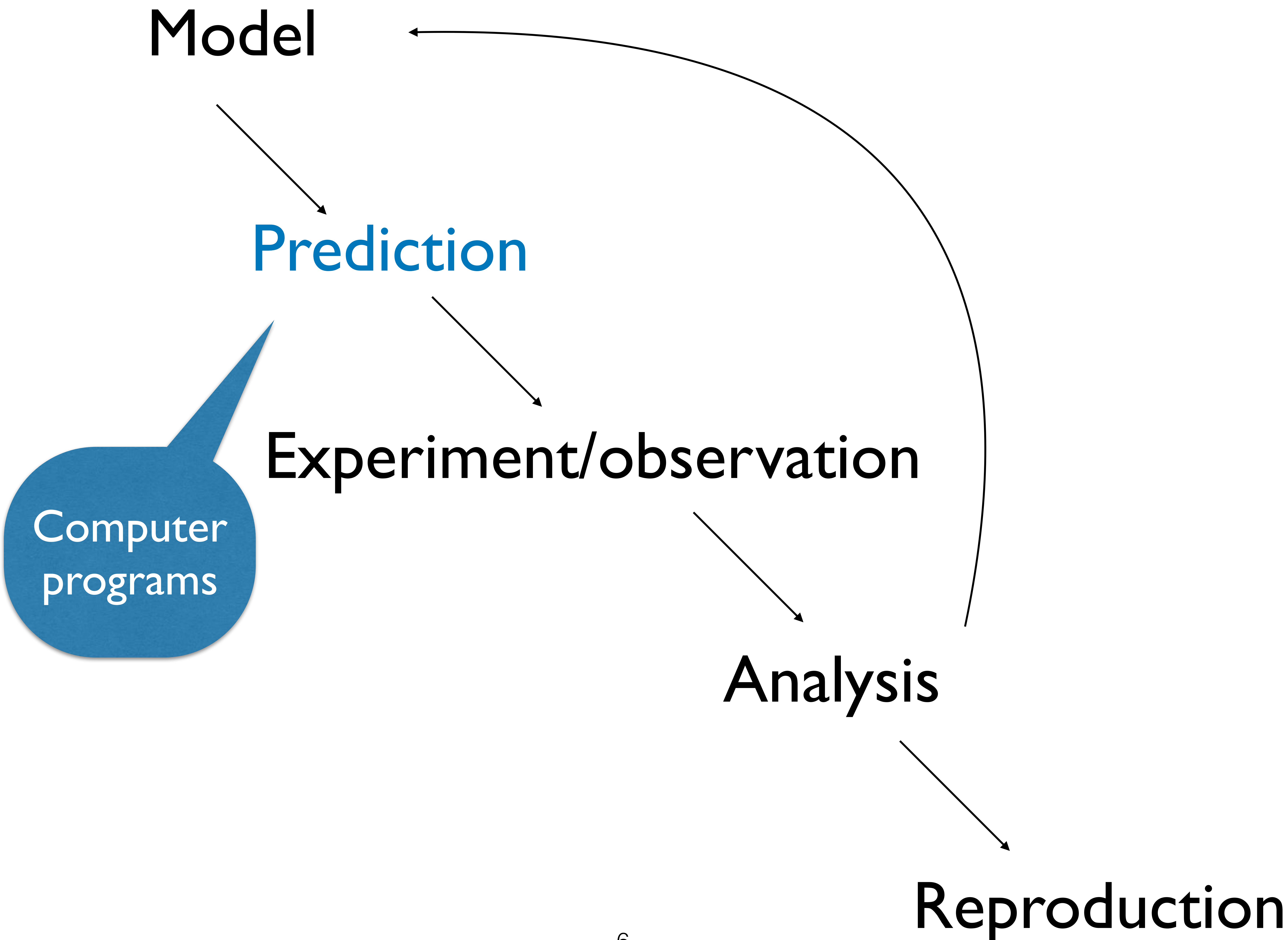
Dominic Orchard[1], Andrew Rice[2]

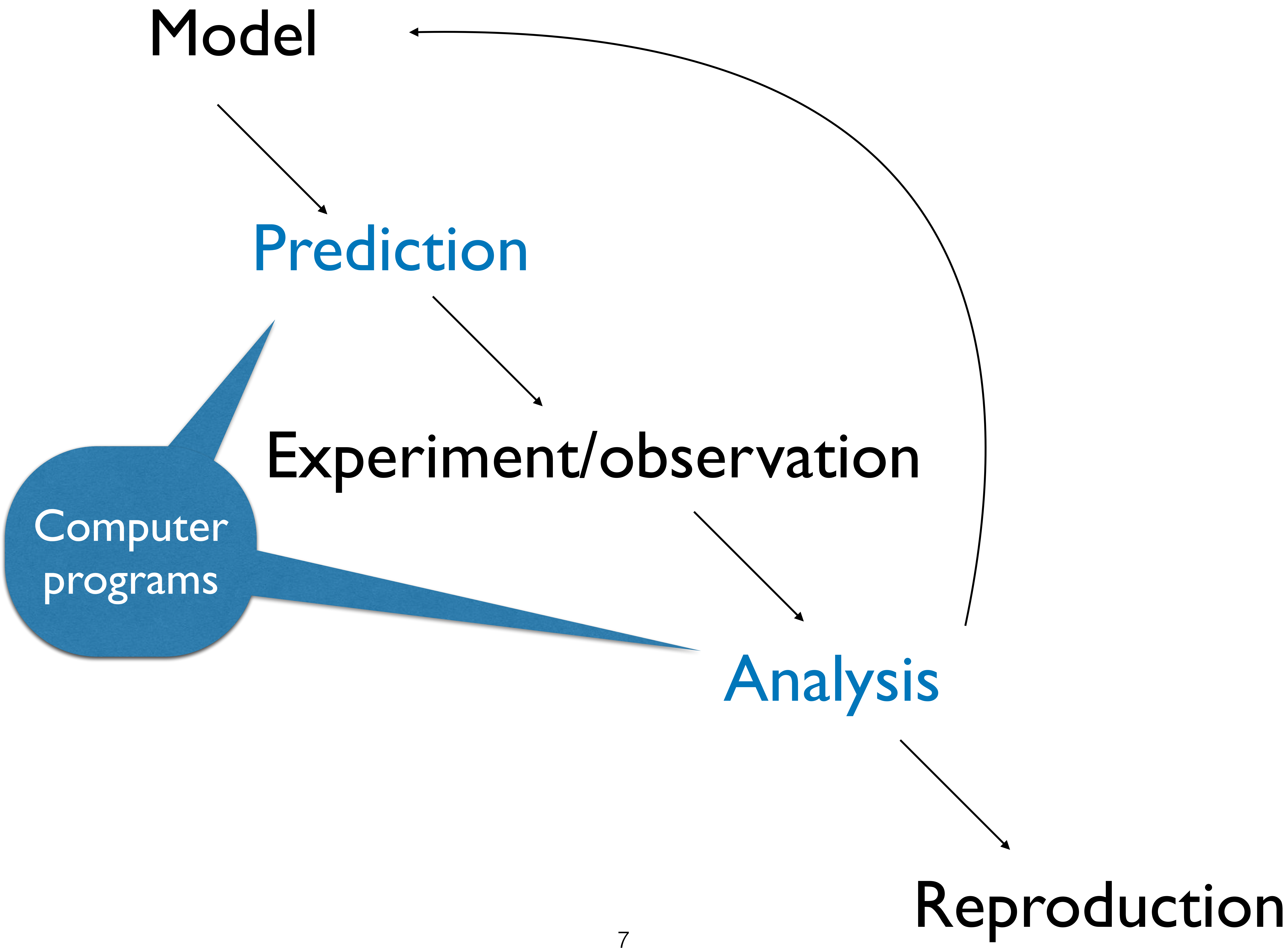
[1] Computer Laboratory, University of Cambridge
dominic.orchard@cl.cam.ac.uk
[2] Computer Laboratory, University of Cambridge
andrew.rice@cl.cam.ac.uk

Abstract
Scientific models are often expressed as large and complicated programs. These programs embody numerous assumptions made by the developer (*e.g.*, for differential equations, the discretization strategy and resolution). The complexity and pervasiveness of these assumptions means that often the only true description of the model is the software itself. This has led various researchers to call for scientists to publish their source code along with their papers. We argue that this is unlikely to be beneficial since it is almost impossible to separate implementation assumptions from the original scientific intent. Instead we advocate higher-level abstractions in programming languages, coupled with lightweight verification techniques such as specification and type systems. In this position paper, we suggest several novel techniques and outline an evolutionary approach to applying these to existing and future models. One-dimensional heat flow is used as an example throughout.

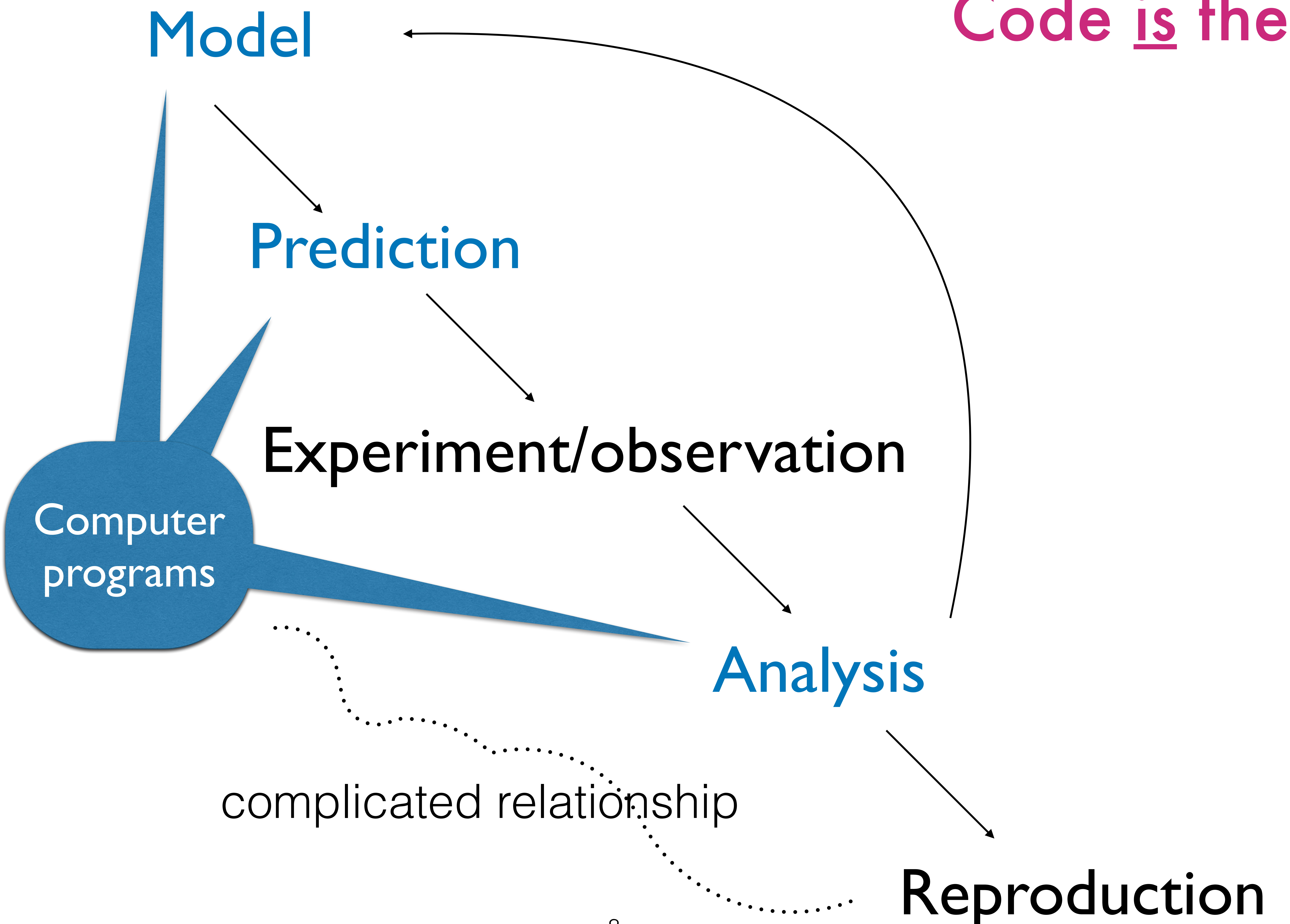
Keywords: computational science, modelling, programming, verification, reproducibility, abstractions, type systems, language design


Model
Prediction
Experiment/observation
Analysis
Reproduction


Model
Prediction
Computer programs
Experiment/observation
Analysis
Reproduction


Model
Prediction
Experiment/observation
Computer programs
Analysis
Reproduction

Code is the model



Example 1D heat equation

Abstract model

$$\frac{\partial \phi}{\partial t} = \alpha \frac{\partial^2 \phi}{\partial x^2}$$

Solution strategy

$$\phi_x^t = \phi_x^{t-1} + \frac{\alpha \Delta t}{\Delta x^2}(\phi_{x+1}^{t-1} + 2\phi_x^{t-1} + \phi_{x-1}^{t-1})$$

Example 1D heat equation

Abstract model

$$\frac{\partial \phi}{\partial t} = \alpha \frac{\partial^2 \phi}{\partial x^2}$$

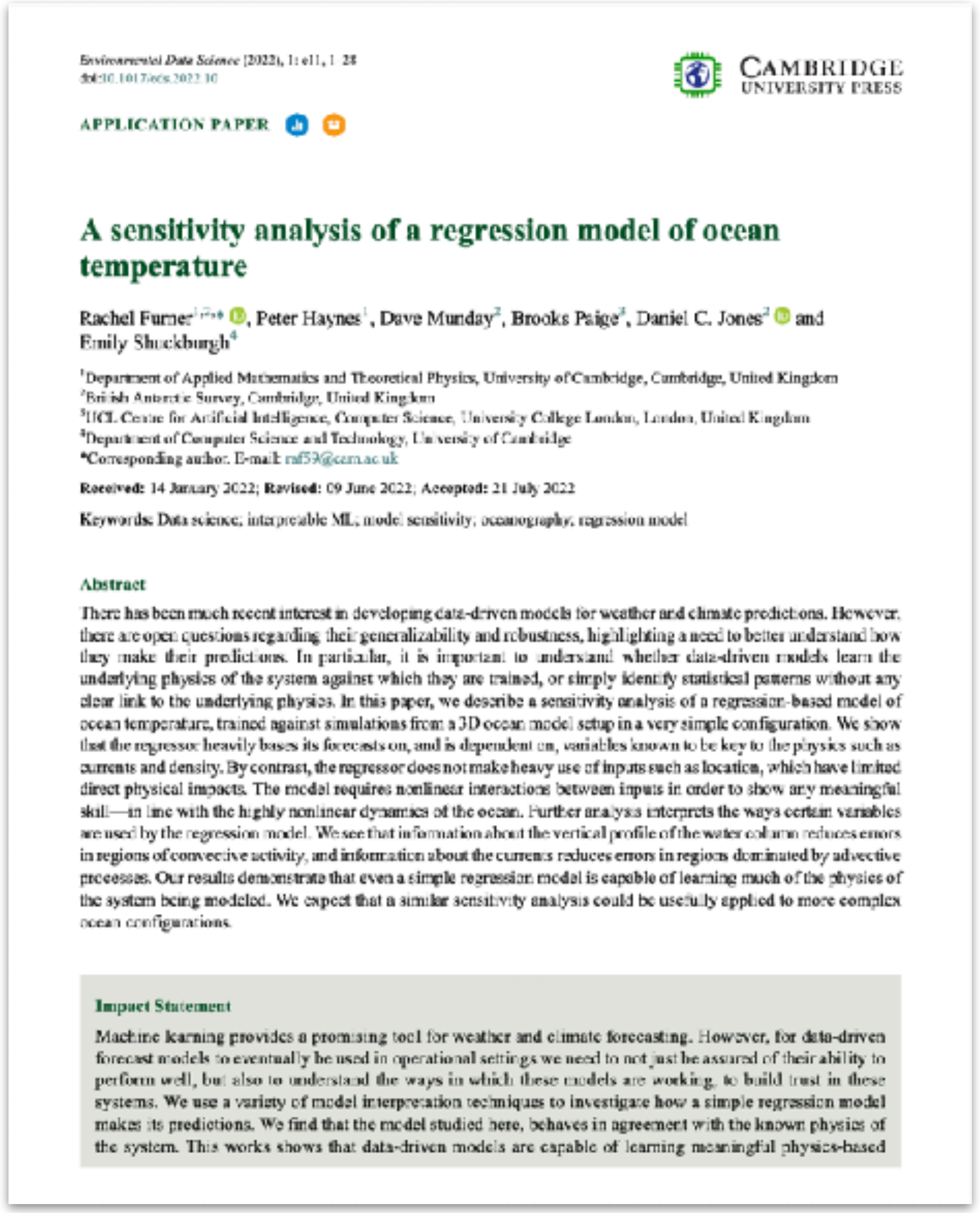
Solution strategy

$$\phi_x^t = \phi_x^{t-1} + \frac{\alpha \Delta t}{\Delta x^2}(\phi_{x+1}^{t-1} + 2\phi_x^{t-1} + \phi_{x-1}^{t-1})$$

Prediction calculation

```
tend = ...              % end time
xmax = ...              % length of material
dt   = ...              % time resolution
dx   = ...              % space resolution
alpha = ...             % diffusion coefficient
nt = tend/dt            % # of time steps
nx = xmax/dx            % # of space steps
r = alpha*dt/dx^2       % constant in solution

real h(0,nx),            % heat fun. (discretised
     h_old(0, nx);       % in space) at t and t-1

do t = 0, nt
  h_old = h
  do x = 1, nx - 1
       h(i) = h_old(i) + r*(h_old(i-1))
                  - 2*h_old(i) + h_old(i+1)
  end do
end do
```

Gap in explanation....


Environmental Data Science (2022), 1: e11, 1–28

CAMBRIDGE UNIVERSITY PRESS

APPLICATION PAPER

A sensitivity analysis of a regression model of ocean temperature

Rachel Furner[1,2,*], Peter Haynes[1], Dave Munday[2], Brooks Paige[3], Daniel C. Jones[2] and Emily Shuckburgh[4]

[1]Department of Applied Mathematics and Theoretical Physics, University of Cambridge, Cambridge, United Kingdom
[2]British Antarctic Survey, Cambridge, United Kingdom
[3]UCL Centre for Artificial Intelligence, Computer Science, University College London, London, United Kingdom
[4]Department of Computer Science and Technology, University of Cambridge
*Corresponding author. E-mail: raf59@cam.ac.uk

Received: 14 January 2022; **Revised:** 09 June 2022; **Accepted:** 21 July 2022

Keywords: Data science; interpretable ML; model sensitivity; oceanography; regression model

Abstract

There has been much recent interest in developing data-driven models for weather and climate predictions. However, there are open questions regarding their generalizability and robustness, highlighting a need to better understand how they make their predictions. In particular, it is important to understand whether data-driven models learn the underlying physics of the system against which they are trained, or simply identify statistical patterns without any clear link to the underlying physics. In this paper, we describe a sensitivity analysis of a regression-based model of ocean temperature, trained against simulations from a 3D ocean model setup in a very simple configuration. We show that the regressor heavily bases its forecasts on, and is dependent on, variables known to be key to the physics such as currents and density. By contrast, the regressor does not make heavy use of inputs such as location, which have limited direct physical impacts. The model requires nonlinear interactions between inputs in order to show any meaningful skill—in line with the highly nonlinear dynamics of the ocean. Further analysis interprets the ways certain variables are used by the regression model. We see that information about the vertical profile of the water column reduces errors in regions of convective activity, and information about the currents reduces errors in regions dominated by advective processes. Our results demonstrate that even a simple regression model is capable of learning much of the physics of the system being modeled. We expect that a similar sensitivity analysis could be usefully applied to more complex ocean configurations.

Impact Statement

Machine learning provides a promising tool for weather and climate forecasting. However, for data-driven forecast models to eventually be used in operational settings we need to not just be assured of their ability to perform well, but also to understand the ways in which these models are working, to build trust in these systems. We use a variety of model interpretation techniques to investigate how a simple regression model makes its predictions. We find that the model studied here, behaves in agreement with the known physics of the system. This works shows that data-driven models are capable of learning meaningful physics-based

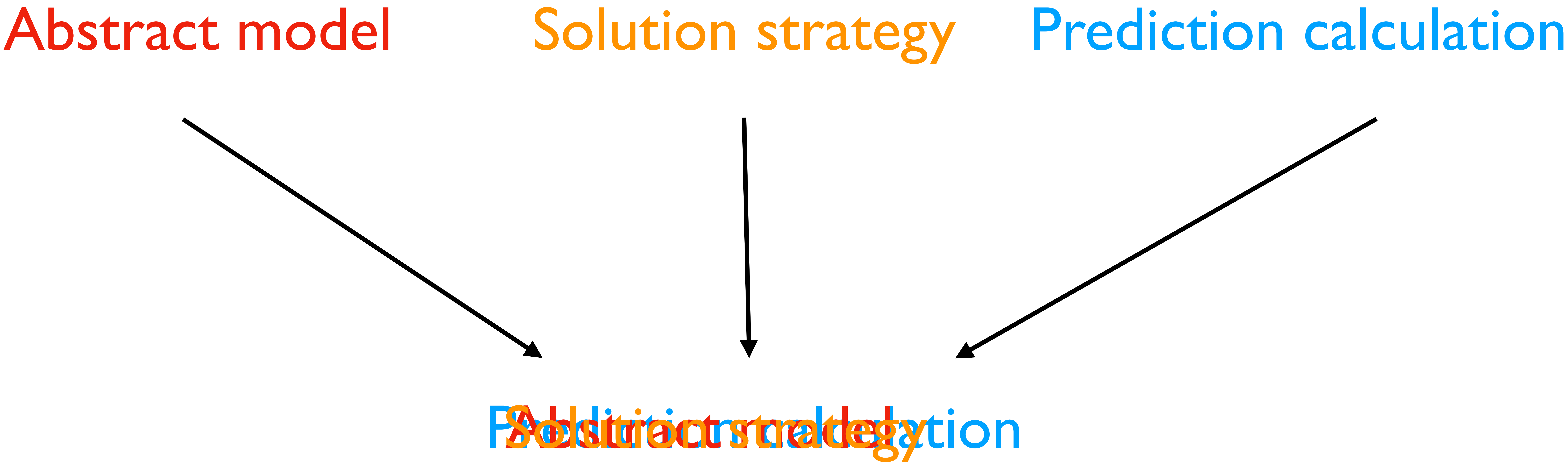
```
module simulation_mod
  use helpers_mod
  implicit none

  contains

  subroutine compute_tentative_velocity(u, v, f, g, flag, del_t)
    real u(0:imax+1, 0:jmax+1), v(0:imax+1, 0:jmax+1), f(0:imax+1, 0:jmax+1),  &
         g(0:imax+1, 0:jmax+1)
    integer flag(0:imax+1, 0:jmax+1)
    real, intent(in) :: del_t

    integer i, j
    real du2dx, duvdy, duvdx, dv2dy, laplu, laplv

    do i = 1, (imax-1)
      do j = 1, jmax
        ! only if both adjacent cells are fluid cells */
        if (toLogical(iand(flag(i,j), C_F)) .and.                            &
            toLogical(iand(flag(i+1,j), C_F))) then

          du2dx = ((u(i,j)+u(i+1,j))*(u(i,j)+u(i+1,j))+                       &
                  gamma*abs(u(i,j)+u(i+1,j))*(u(i,j)-u(i+1,j))-               &
                  (u(i-1,j)+u(i,j))*(u(i-1,j)+u(i,j))-                        &
                  gamma*abs(u(i-1,j)+u(i,j))*(u(i-1,j)-u(i,j)))               &
                  /(4.0*delx)
          duvdy = ((v(i,j)+v(i+1,j))*(u(i,j)+u(i,j+1))+                       &
                  gamma*abs(v(i,j)+v(i+1,j))*(u(i,j)-u(i,j+1))-               &
                  (v(i,j-1)+v(i+1,j-1))*(u(i,j-1)+u(i,j))-                    &
                  gamma*abs(v(i,j-1)+v(i+1,j-1))*(u(i,j-1)-u(i,j)))           &
                     /(4.0*dely)
          laplu = (u(i+1,j)-2.0*u(i,j)+u(i-1,j))/delx/delx+                  &
                  (u(i,j+1)-2.0*u(i,j)+u(i,j-1))/dely/dely

          f(i,j) = u(i,j) + del_t*(laplu/Re-du2dx-duvdy)
        else
          f(i,j) = u(i,j)
        end if
      end do
    end do
```

Abstract model

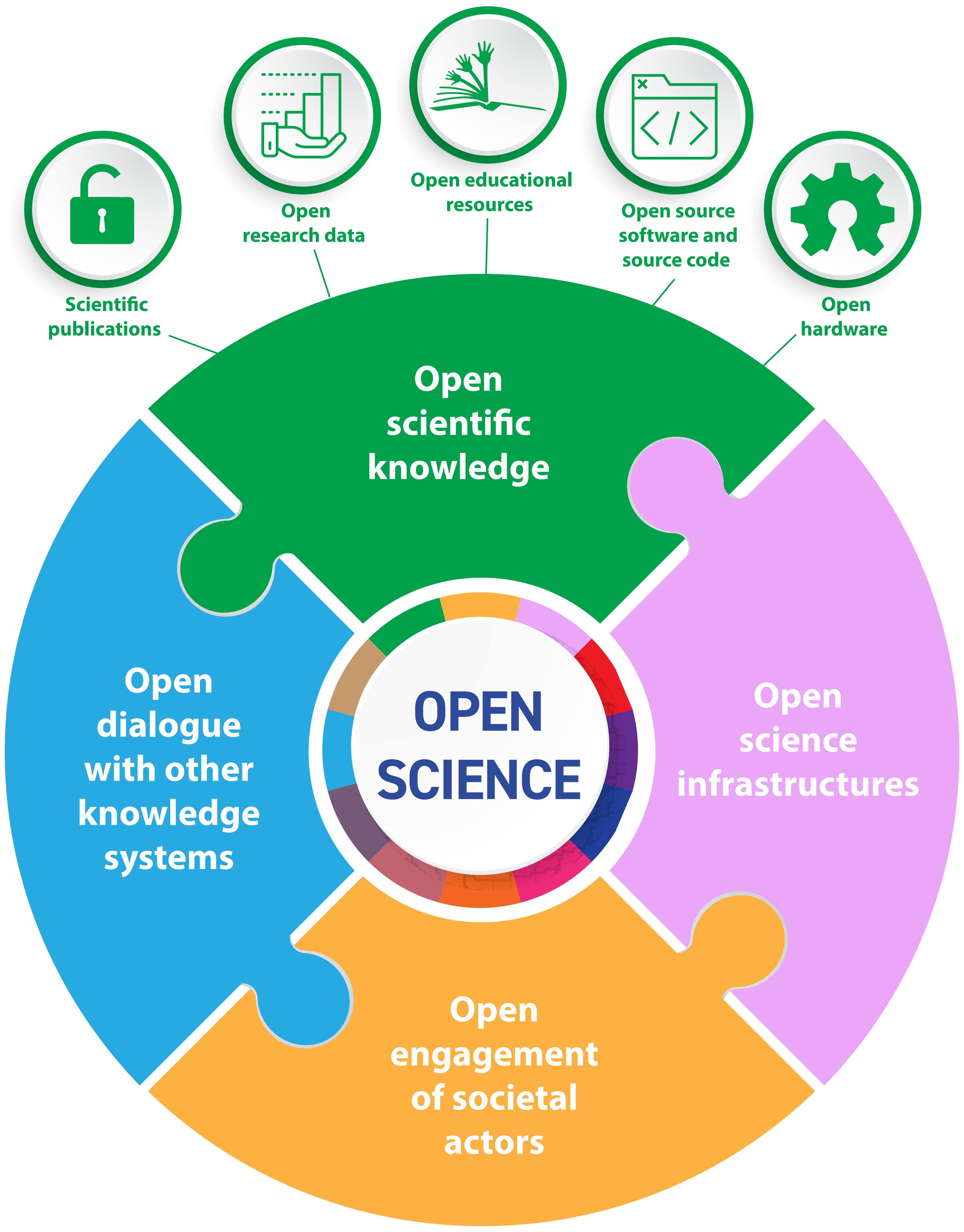
Solution strategy

Prediction calculation

Conflation of concerns



Code conflates & hides many aspects of the model



UNESCO 2021 Open Science recommendation


Open research data
Open educational resources
Open source software and source code
Open hardware
Open scientific

But.. sharing code includes sharing bugs

+ assumptions
+ incidental decisions
+ approximations

Open problem: separating and relating concerns

Abstract model — Solution strategy — Prediction calculation

Partial solutions

- Extra technical documentation
- Clear systems design
- High modularity

Could there be better support via a **programming language tailored to science?**

BETTER
LANGUAGES
BETTER
SOFTWARE
BETTER
RESEARCH

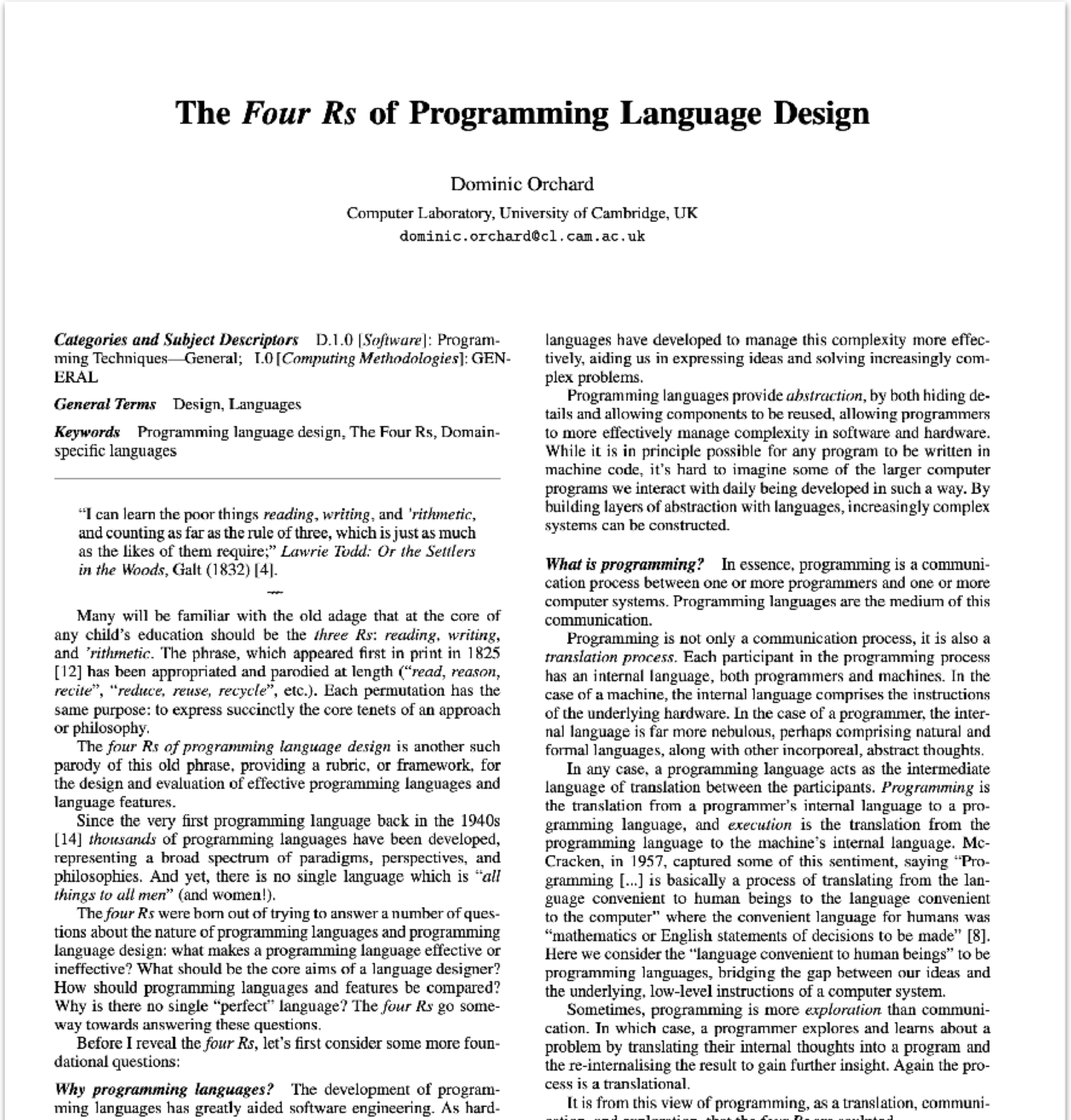
The four Rs of programming language design... (Orchard, 2011)

Reading

w**R**iting

Reasoning

Running



The *Four Rs* of Programming Language Design

Dominic Orchard
Computer Laboratory, University of Cambridge, UK
dominic.orchard@cl.cam.ac.uk

Categories and Subject Descriptors D.1.0 [*Software*]: Programming Techniques—General; I.0 [*Computing Methodologies*]: GENERAL

General Terms Design, Languages

Keywords Programming language design, The Four Rs, Domain-specific languages

"I can learn the poor things *reading*, *writing*, and *'rithmetic*, and counting as far as the rule of three, which is just as much as the likes of them require;" *Lawrie Todd: Or the Settlers in the Woods*, Galt (1832) [4].

Many will be familiar with the old adage that at the core of any child's education should be the *three Rs*: *reading*, *writing*, and *'rithmetic*. The phrase, which appeared first in print in 1825 [12] has been appropriated and parodied at length ("*read, reason, recite*", "*reduce, reuse, recycle*", etc.). Each permutation has the same purpose: to express succinctly the core tenets of an approach or philosophy.

The *four Rs of programming language design* is another such parody of this old phrase, providing a rubric, or framework, for the design and evaluation of effective programming languages and language features.

Since the very first programming language back in the 1940s [14] *thousands* of programming languages have been developed, representing a broad spectrum of paradigms, perspectives, and philosophies. And yet, there is no single language which is "*all things to all men*" (and women!).

The *four Rs* were born out of trying to answer a number of questions about the nature of programming languages and programming language design: what makes a programming language effective or ineffective? What should be the core aims of a language designer? How should programming languages and features be compared? Why is there no single "perfect" language? The *four Rs* go someway towards answering these questions.

Before I reveal the *four Rs*, let's first consider some more foundational questions:

Why programming languages? The development of programming languages has greatly aided software engineering. As hardware has developed to manage this complexity more effectively, aiding us in expressing ideas and solving increasingly complex problems.

Programming languages provide *abstraction*, by both hiding details and allowing components to be reused, allowing programmers to more effectively manage complexity in software and hardware. While it is in principle possible for any program to be written in machine code, it's hard to imagine some of the larger computer programs we interact with daily being developed in such a way. By building layers of abstraction with languages, increasingly complex systems can be constructed.

What is programming? In essence, programming is a communication process between one or more programmers and one or more computer systems. Programming languages are the medium of this communication.

Programming is not only a communication process, it is also a *translation process*. Each participant in the programming process has an internal language, both programmers and machines. In the case of a machine, the internal language comprises the instructions of the underlying hardware. In the case of a programmer, the internal language is far more nebulous, perhaps comprising natural and formal languages, along with other incorporeal, abstract thoughts.

In any case, a programming language acts as the intermediate language of translation between the participants. *Programming* is the translation from a programmer's internal language to a programming language, and *execution* is the translation from the programming language to the machine's internal language. McCracken, in 1957, captured some of this sentiment, saying "Programming [...] is basically a process of translating from the language convenient to human beings to the language convenient to the computer" where the convenient language for humans was "mathematics or English statements of decisions to be made" [8]. Here we consider the "language convenient to human beings" to be programming languages, bridging the gap between our ideas and the underlying, low-level instructions of a computer system.

Sometimes, programming is more *exploration* than communication. In which case, a programmer explores and learns about a problem by translating their internal thoughts into a program and the re-internalising the result to gain further insight. Again the process is a translational.

It is from this view of programming, as a translation, communi-

The Two Complexities

Inherent

Inadequately supported

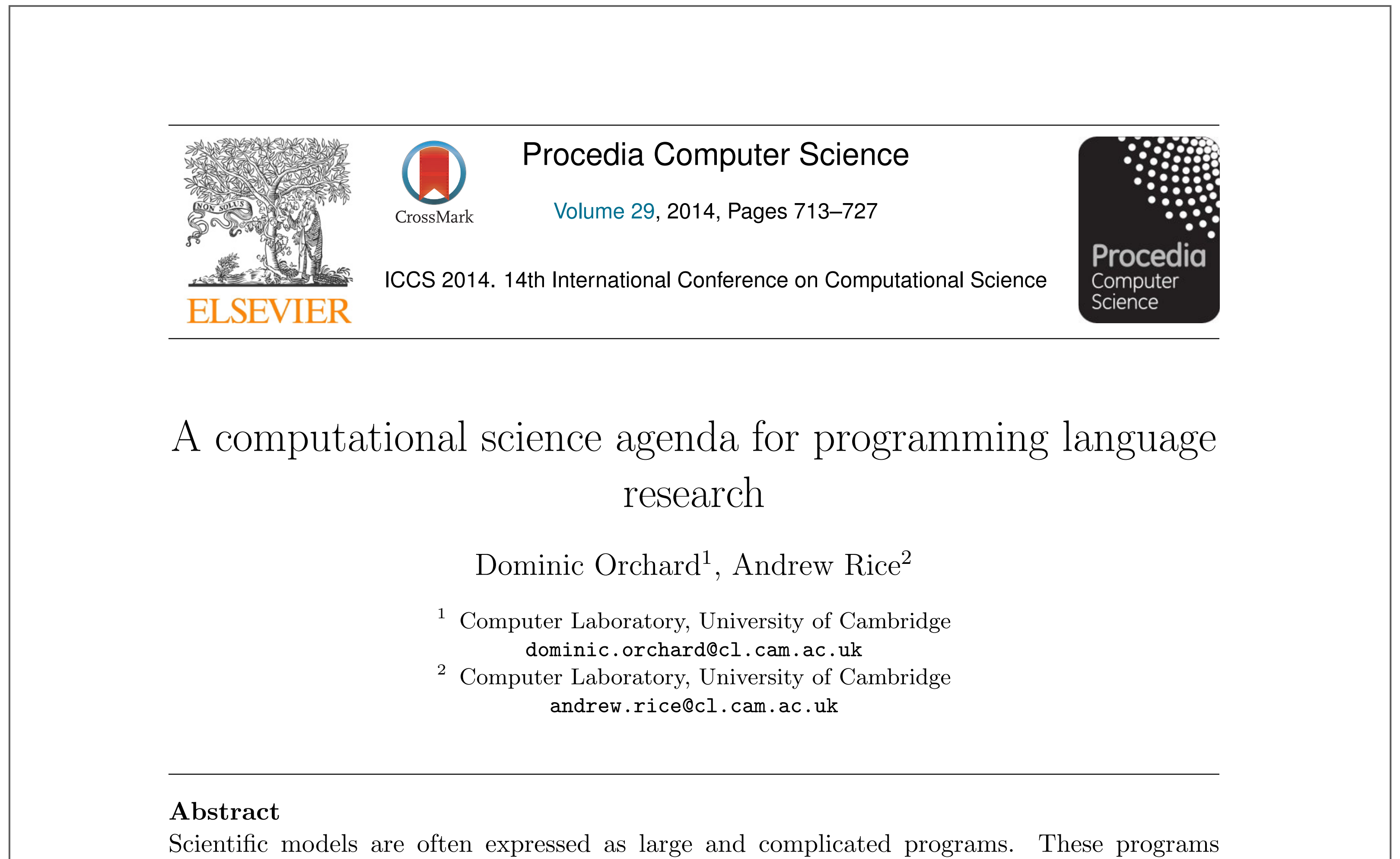
Accidental

Too easy to introduce

Both hinder scientific progress, only one is necessary

Roadmap

1. Computer science engagement with scientists
2. New systems for abstraction and specification
3. Evolutionary approach for languages



Procedia Computer Science

Volume 29, 2014, Pages 713–727

ICCS 2014. 14th International Conference on Computational Science

ELSEVIER

CrossMark

Procedia Computer Science

A computational science agenda for programming language research

Dominic Orchard[1], Andrew Rice[2]

[1] Computer Laboratory, University of Cambridge
dominic.orchard@cl.cam.ac.uk
[2] Computer Laboratory, University of Cambridge
andrew.rice@cl.cam.ac.uk

Abstract
Scientific models are often expressed as large and complicated programs. These programs

Validation

Did we implement the right equations?

vs

Verification

Did we implement the equations right?

Challenge

Telling these two apart when results are not as expected

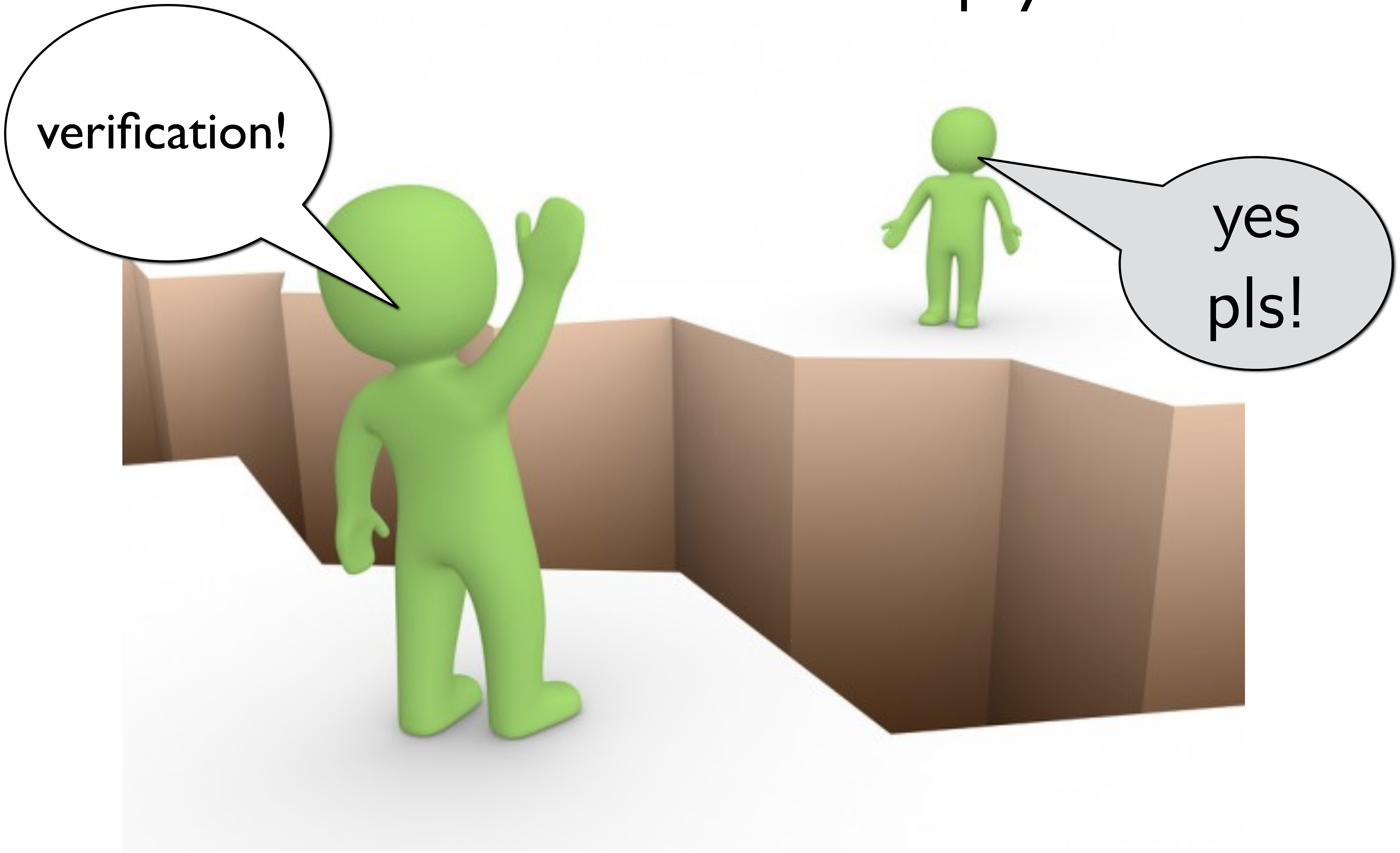
Software bugs undermine reproduction

Testing	vs.	Verification
"Smoke" testing		Types
Unit testing		Static analysis
Integration testing		Specify-and-check

Important, but incomplete (does not rule out all bugs)

Largely unexplored in climate science

natural & physical sciences



computer science

natural & physical sciences

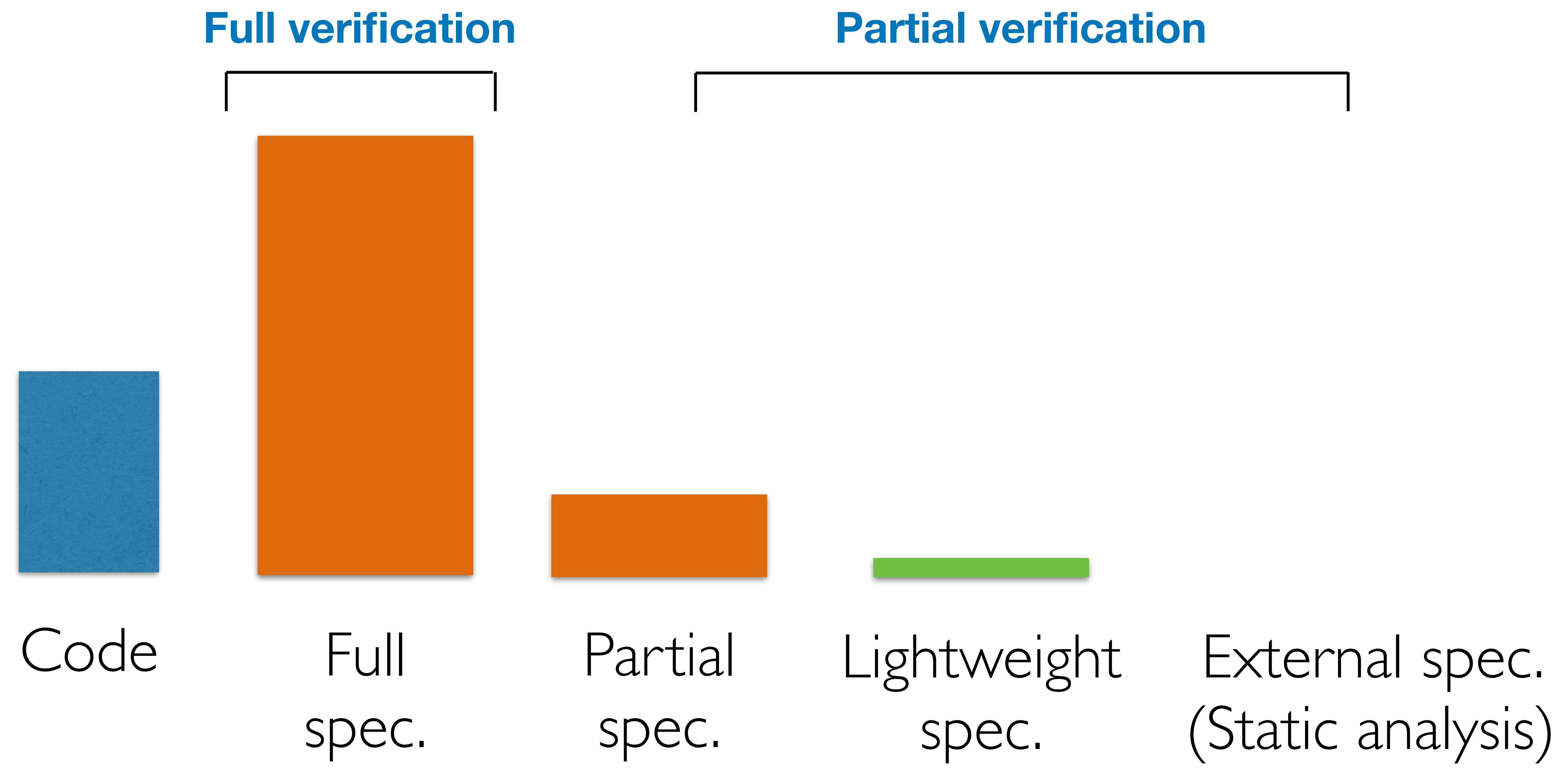
computer science

natural & physical sciences

computer science

Let's bridge the chasm!

Approaches to verification



Time consuming
Specification completeness?

How to chose which parts?

Focussed on one aspect

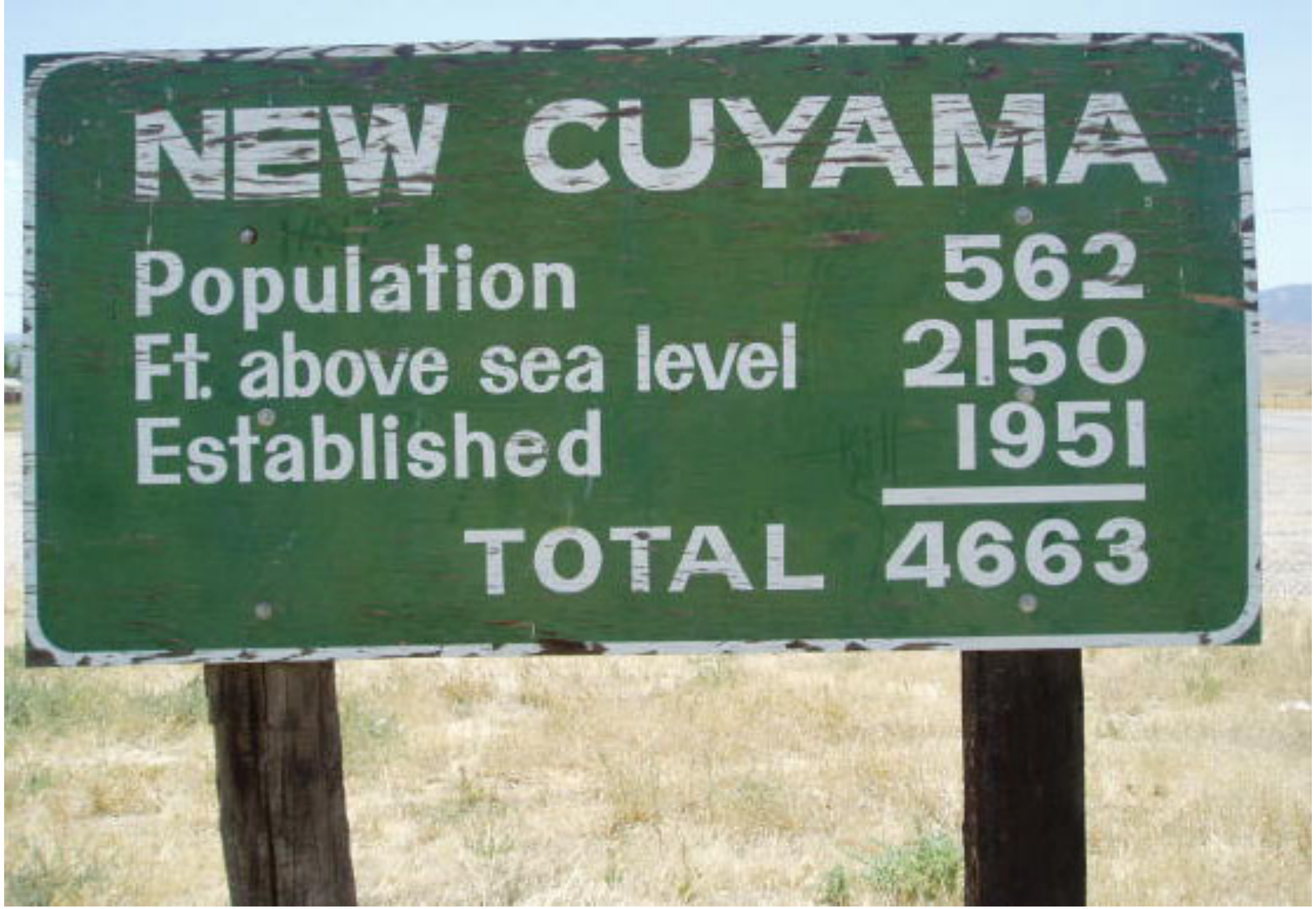


photo from Andrew Kennedy's website
http://research.microsoft.com/en-us/um/people/akenn/units/

Journal of Computational Science 9 (2015) 156–162

Contents lists available at ScienceDirect

Journal of Computational Science

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

Evolving Fortran types with inferred units-of-measure

Dominic Orchard [a,*], Andrew Rice [b], Oleg Oshmyan [b]

[a] Department of Computing, Imperial College London, United Kingdom
[b] Computer Laboratory, University of Cambridge, United Kingdom

ARTICLE INFO

Article history:
Available online 18 April 2015

Keywords:
Units-of-measure
Dimension typing
Type systems
Verification
Code base evolution
Fortran
Language design

ABSTRACT

Dimensional analysis is a well known technique for checking the consistency of equations involving physical quantities, constituting a kind of type system. Various type systems for dimensional analysis, and its refinement to units-of-measure, have been proposed. In this paper, we detail the design and implementation of a units-of-measure system for Fortran, provided as a pre-processor. Our system is designed to aid adding units to existing code base: units may be polymorphic and can be inferred. Furthermore, we introduce a technique for reporting to the user a set of *critical variables* which should be explicitly annotated with units to get the maximum amount of unit information with the minimal number of explicit declarations. This aids adoption of our type system to existing code bases, of which there are many in computational science projects.

1. Introduction

Type systems are one of the most popular static techniques for recognizing and rejecting large classes of programming error. A common analogy for types is of physical quantities (*e.g.*, in [2]), where type checking excludes, for example, the non-sensical addition of non-comparable quantities such as adding 3 m to 2 J; they have different *dimensions* (length *vs.* energy) and different *units* (metres *vs.* joules). This analogy between types and dimensions/units goes deeper. The approach of *dimensional analysis* checks the consistency of formulae involving physical quantities, acting as a kind of type system (performed by hand, long before computers). Various automatic type-system-like approaches have been proposed for including dimensional analysis in programming languages (*e.g.* [10] is a famous paper detailing one such approach, which also cites much of the relevant history of other systems).

Failing to ensure that the dimensions (or units) of values are correctly matched can be disastrous. An extreme example of this is the uncaught unit mismatch which led to the destruction of the

circumstances. It therefore seems inevitable that these errors are likely in computational science too.

The importance of units is often directly acknowledged in source code. We have seen source files carefully commented with the units and dimensions of each variable and parameter. We have also watched programmers trying to use this information: a process of scrolling up and down, repeatedly referring to the unit specification of each parameter. Incorporating units into the type system would move the onus of responsibility from the programmer to the compiler.

A recent ISO standards proposal (N1969) for Fortran introduces a units-of-measure system which follows Fortran's tradition of explicitness [7]. Every variable declaration must have an explicit unit declaration and every composite unit (*e.g.*, metres times seconds) must itself be explicitly declared. This imposes the extra burden of annotating variables directly on the programmer. As an example, we studied two medium-sized models (roughly 10,000 lines of code each) and found roughly a 1:10 ratio between variable declarations and lines of code. Thus, adding explicit units of

Editors: Konrad Hinsen, konrad.hinsen@cnrs-orleans.fr | Matthew Turk, matthewturk@gmail.com

Units-of-Measure Correctness in Fortran Programs

Mistral Contrastin, Andrew Rice, and Matthew Danish | University of Cambridge
Dominic Orchard | Imperial College London

Much of mathematics' use in science revolves around measurements of physical quantities, both abstractly and concretely. Such measurements are naturally classified by their *dimension*, that is, whether the measurement is of distance, energy, time, and so on. Dimensionality is further refined by a measurement's units-of-measure (or units, for short), such as

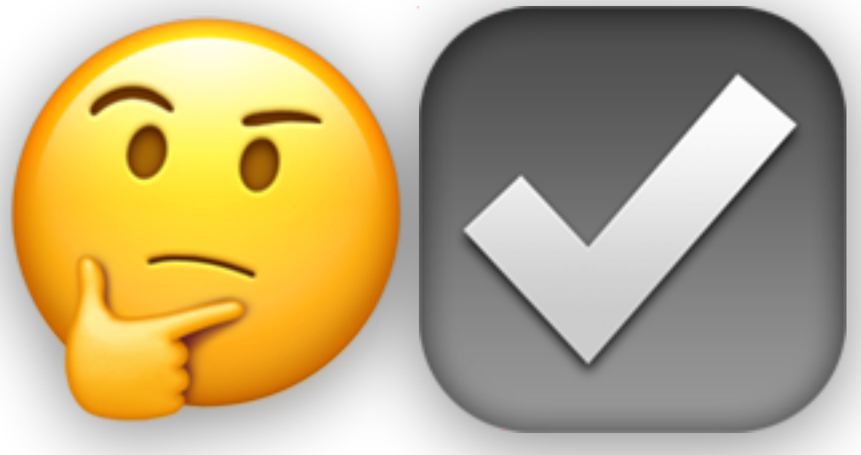
lightweight, nonbinding specification and analysis too can help find bugs in programs before they strike. We that, in general, these kinds of program analysis tools w come more widely used by scientists to save time and r grief during the development process, as well as inc confidence in results of numerical models.

Ensuring the consistent use of units is an impo

CamFort

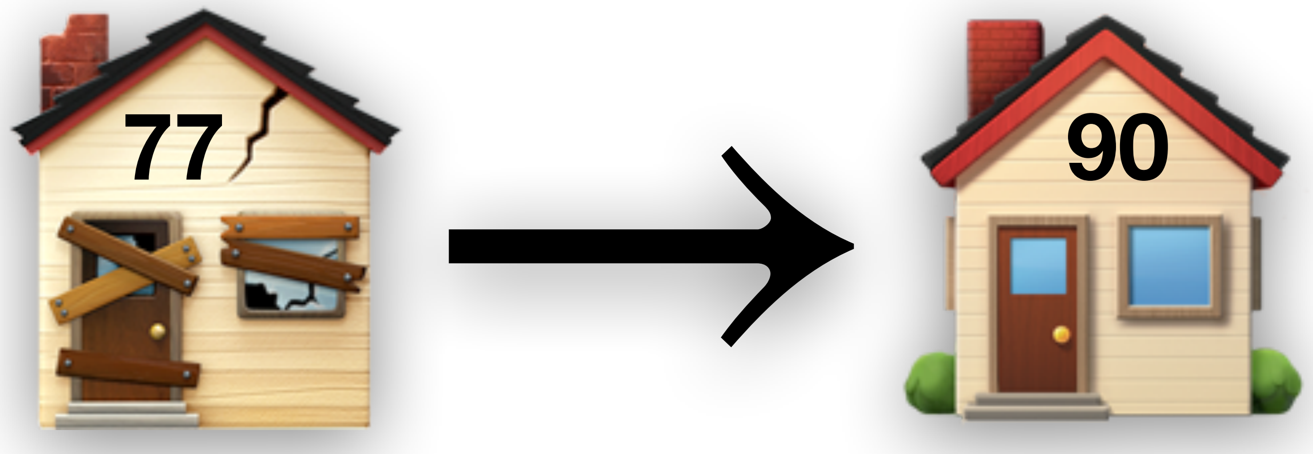
https://github.com/camfort/camfort/

Verification



Analysis

Refactoring



UKRI Engineering and Physical Sciences Research Council

Met Office Hadley Centre

Bloomberg

Units-of-measure verification

```
1   program energy
2     real :: mass = 3.00, gravity = 9.91, height = 4.20
3     real :: potential_energy
4
5     potential_energy = mass * gravity * height
6   end program energy
```

Suggest

```
$ camfort units-suggest energy1.f90
```

```
Suggesting variables to annotate with unit specifications in 'energy1.f90'
…
energy1.f90: 3 variable declarations suggested to be given a
specification:
    energy1.f90 (2:43)    height
    energy1.f90 (2:14)    mass
    energy1.f90 (3:14)    potential_energy
```

Units-of-measure verification

```fortran
program energy
  != unit kg :: mass
  != unit m  :: height
  real :: mass = 3.00, gravity = 9.91, height = 4.20
  != unit kg m**2/s**2 :: potential_energy
  real :: potential_energy

  potential_energy = mass * gravity * height
end program energy
```

Check

```
$ camfort units-check energy1.f90

energy1.f90: Consistent. 4 variables checked.
```

Units-of-measure verification

```
program energy
  != unit kg :: mass
  != unit m  :: height
  real :: mass = 3.00, gravity = 9.91, height = 4.20
  != unit kg m**2/s**2 :: potential_energy
  real :: potential_energy

  potential_energy = mass * gravity * height
end program energy
```

Synthesise

```
$ camfort units-synth energy1.f90 energy1.f90

Synthesising units for energy1.f90
```

Units-of-measure verification

```
program energy
  != unit kg :: mass
  != unit m  :: height
  != unit m/s**2  :: gravity
  real :: mass = 3.00, gravity = 9.91, height = 4.20
  != unit kg m**2/s**2 :: potential_energy
  real :: potential_energy

  potential_energy = mass * gravity * height
end program energy
```

Synthesise

```
$ camfort units-synth energy1.f90 energy1.f90

Synthesising units for energy1.f90
```

Check

Does it do what I think it does?

Infer

What does it do?

Synthesise

Capture what it does for documentation & future-proofing

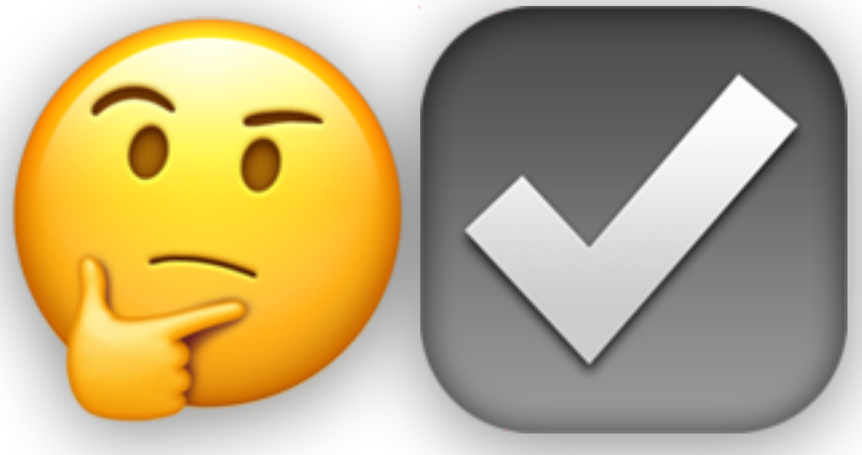
Suggest

Where should I add a specification to get the most information?

Analysis

CamFort

Verification



Java

See general tools e.g.

sonar

coverity

Meta

Infer

Take home messages

- Are we done with language developments? No! But changes slow
- Verification for:
  - Increasing trust
  - Speeding up development
  - Enabling reuse
- I am keen to explore more ideas in this space

@dorchard

https://dorchard.github.io

https://camfort.github.io