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

1 Introduction

With the increase in computer modelling in the sciences, programming languages are now an important tool for expressing complex scientific theories. However, this use of computer models has not changed the fundamental scientific method of *hypothesis*, *prediction*, *experiment*, *analysis*, and *reproduction* [Var10]. Despite this immutability of the scientific method, computer modelling has substantially increased the complexity of both *prediction* and *reproduction*. For example, one might imagine the method applied to the question of one-dimensional heat flow:

1. **Hypothesis** A researcher argues that the change in heat within an object can be related to time and space by a particular (second-order) differential equation.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.064

2. **Prediction** They implement a computer model which finds solutions to this equation for particular initial conditions and parameters. A complex hypothesis with many factors requires numerous design decisions (such as choice of numerical approximation algorithm).

3. **Experiment** They design an experiment involving heating a length of wire (perhaps) and making temperature measurements at particular locations and times.

4. **Analysis** The experimental results are compared to the prediction.

5. **Reproduction** An independent researcher considers the whole process: does the prediction and model follow from the hypothesis? does the model correctly predict the results of new experiments? do different approximation approaches give the same conclusions?

Particular challenges in the implementation of the model (the prediction step) are to capture all the aspects of the hypothesis and produce an implementation which expresses this intent without bugs, false assumptions, or over-approximations. This becomes increasingly difficult as the complexity of the hypothesis increases. Any researcher attempting to reproduce these results must tackle these challenges again to generate their own predictions. Sharing the source code of the model does little to help since this also shares any implicit assumptions, over-approximations, and bugs. Thus, by reproducibility we mean the ability to independently valid a theory, rather than to replicate results by rerunning a program (see discussion in [Dru09]).

In addition to the *essential complexity* introduced by a complex hypothesis, computer models also suffer from *accidental complexity*. Accidental complexity can arise from many sources such as complicated programming language syntax or performance optimisations introduced by programmers. Programming language research attempts to reduce these complexities by developing better tools for programmers. In this position paper, we focus on the benefit of high-level abstractions and specifications. These provide researchers with confidence in the correctness of their program and guarantee the correctness of specifications shared with others.

The `forall` construct of Fortran 95 (originating in High Performance Fortran [KKZ07]) is an example of a beneficial language abstraction. It abstracts a loop over an array's domain but restricts the programmer such that the loop body has no side effects or output dependencies. This aids verification and performance (the `forall` body is data parallel). We believe that much higher levels of abstraction are possible.

We introduce first a simple example using one-dimensional heat flow (Section 2.1) which we use to draw attention to issues of correctness (Section 2.2) and code quality (Section 2.3). To gain adoption in computational science, any potential solutions to these issues must also integrate with existing software and working practices and so we envisage an evolutionary approach beginning with existing programming languages. We have begun implementing some examples of this in our free, open-source Fortran refactoring tool *CamFort* [OR13]. We give an overview of this evolutionary approach (Section 3) and illustrate it with transformations of our heat flow example (Sections 3.1 and 3.2). Along with this, we propose programming concepts for further research, which address the issues of Section 2.

These ideas have developed from our work with climate science researchers and we now hope to engage the wider community in identifying opportunities and areas to investigate further.

2 Key issues in programming for computational science

We have seen two major classes of problem corresponding to *correctness*: either that of the implementation or the conceptual model itself, and *code quality*: either in communicating

```
1  integer            :: t, x
2  real, parameter    :: alpha = ...           ! diffusion coefficient
3  real, parameter    :: tmax = ...            ! end time
4  real, parameter    :: xmax = ...            ! length of material
5  real, parameter    :: dt   = ...            ! time resolution
6  real, parameter    :: dx   = ...            ! space resolution
7  integer, parameter :: nt = tmax/dt          ! # of time steps
8  integer, parameter :: nx = xmax/dx          ! # of space steps
9  real, parameter    :: r = alpha*dt/(dx*dx)  ! constant in solution
10
11 real :: h(1:nx), h_old(1:nx)   ! heat function (discretised in space) at t and t-1
12 h    = 0                      ! initialise as cold
13 h(1) = 1                      !   with one hot end
14 do t = 1, nt
15    h_old = h
16    forall (x = 2:(nx - 1))  h(x) = h_old(x) + r*(h_old(x-1) - 2*h_old(x) + h_old(x+1))
17 end do
```

Figure 1: Fortran program fragment approximating the one-dimensional heat equation

choices made by the programmer or in the possibility of reusing or extending the software. These are best illustrated in practice, for which we use our introductory one-dimensional heat flow example as a case study throughout.

2.1 Case study: heat equation

The one-dimensional heat equation describes the rate of change of heat in a uniform medium. In abstract form, the relationship between change in heat (energy) in time and space is specified:

$$\frac{\partial \phi}{\partial t} = \alpha \frac{\partial^2 \phi}{\partial x^2} \tag{1}$$

where ϕ is the energy function ($\phi : \mathit{space} \times \mathit{time} \to \mathit{energy}$) and α is the constant diffusion rate. Using standard analytical techniques (*e.g.*, *Taylor series*, *mean value theorem*), the following provides a discrete approximation to the continuous form using a finite-difference scheme with *"Forward Time, Centered Space"* (FTCS) (see [DDD91, Rec04]):

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

where ϕ_x^t is the energy at the discrete time position t and location x, and Δt and Δx are the discrete time and space steps. Figure 1 shows a Fortran implementation calculating ϕ_x^t at time `tmax`. Lines 11-13 define the discrete heat functions ϕ^t and ϕ^{t-1} as `h` and `h_old`. The outer loop (line 14-) iterates over the discretised time dimension and the inner loop, written using `forall` (line 16), iterates over the discretised spatial dimension, computing equation (2). The outer loop cannot be expressed using `forall` since it includes a stateful operation (line 15), updating `h_old` with the values in `h` from the previous temporal iteration.

2.2 Questions of correctness: verification and validation

Programming errors are a well known problem in computational science (and programming in general) and can have a significant negative impact. In one case, a research group in structural biology retracted five papers (three published in *Science*) due to a single programming error

which invalidated their results [CRR+06, Mer10]. As predictions from models are increasingly used to direct social and environmental policy, correctness is not just an abstract concern for researchers. High-assurance models are also important for public confidence in science-led decision making (see, *e.g.*, the "ClimateGate" incident where leaked e-mails showing scientists' own lack of confidence in their programming sparked a credibility debate [Mer10]).

Model *verification* is the process of checking that an implementation is free from programming errors. This is a necessary step before model *validation*– the process of checking a model's correspondence to empirical results [OR10]. Without proper verification, it may be unclear whether incorrect results are due to an invalid model or program errors, or both. Even worse are results which appear correct due to program errors compensating for an invalid model.

Example sources of error Our example contains a variety of error-prone constructions. Firstly, the various indexing operations are a common source of low-level error, such as incorrectly specifying iteration spaces (*e.g.*, `1:nx` instead of `2:(nx-1)`, line 16). Notably in Fortran, iteration spaces cannot be easily abstracted thus there is no way to reduce the number of points where these errors could be introduced.

The risk of such errors grows with programs involving multiple, multi-dimensional arrays, such as in this fragment from a Navier-Stokes fluid simulator (based on Griebel *et al.* [GDN97]):

```
duvdy = ((v(i,j)+v(i+1,j))*(u(i,j)+u(i,j+1)) + gam*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)) - gam*abs(v(i,j-1)+v(i+1,j-1))*(u(i,j-1)-u(i,j)))
```

Such code is at risk from errors such as permuting index or array variable names (*e.g.*, writing `i` instead of `j`) or using the wrong offset (*e.g.*, `i-1` instead of `i+1`). These kinds of error may manifest as obviously incorrect results (if the data can be easily observed and analysed) but sometimes the effect is more subtle.

Secondly, the use of state is prone to error. For example, exchanging the heat function in the previous time step for the current (line 15) is a stateful computation; if instead `h = h_old` is written, then the calculation stays in the initial state. Such problems worsen with the use of parallelism and as implementations grow in size.

Our example program also exploits implicit truncation of reals to integers (line 7-8) and assumptions about the relative magnitude of time and space step-sizes (line 9), both of which are possible sources of data error.

Approaches to verification The need for more rigorous verification strategies in science is well known [OR10]. Usually the responsibility is placed on the scientist but extra support is needed from our languages and tools since "diligence and alertness" are not enough [PV05]. Programming language research has had success in reducing programmer responsibility by automatically preventing various errors at compile time. For example, type systems has been used to aid program verification by recognising and rejecting large classes of programming errors, see *e.g.*, [She05, SV07]. These techniques are relatively simple from a user perspective, compared with more heavyweight approaches using automated theorem provers and proof assistants (*e.g.*, *Coq*). Functional programming languages currently lead the way in type system expressivity.

Many languages (C, Fortran, Java) used by scientists already benefit from some use of types as a simple check on a programs consistency (*e.g.*, multiplying two strings is a type error). These type systems give a basic specification of what is computed by a program (*e.g.*, a floating-point number from floating-point parameters). Some more advanced ideas from functional programming have been adopted in object-oriented languages (for example, the addition of

generics in Java, providing *parametric polymorphism*). We later outline how other ideas from type theory might be incorporated to tackle the specific needs of scientific code.

Since many languages do not have a formal semantics, it is difficult to formally prove larger correctness results. However, verification techniques can be combined with testing to increase confidence. There has been research in the programming language community on automatically generating tests, reducing the burden on the programmer, for example QuickCheck [CH00]. Again, these techniques have not yet crossed over into computational science.

2.3 Conflating concerns and over-commitment

Many computer models consist of three conceptual aspects:

1. an abstract mathematical model (*e.g.*, a system of partial differential equations);
2. a solution strategy (*e.g.*, discretisation and approximation);
3. an implementation of this strategy (*e.g.*, concrete, iterative code over arrays).

In many instances these three aspects are heavily intertwined. This obscures the original intent of the model/scientific theory, which becomes distributed throughout a program and obfuscated by implementation assumptions and choices. This hinders understanding, refinement, validation, and verification (where invalid models and programming errors are indistinguishable).

Our heat flow program is highly specialised to the dimensionality of the problem, the solution method, boundary values, and the topology and size of a particular instance. There are several choices about initialisation and boundaries (lines 12-13, 16), with a *Dirichlet* boundary either end of the array's extent (1 at `h(1)` and 0 at `h(nx)`). It does not easily extend to multiple-dimensions, *e.g.*, to the two-dimensional heat equations, without rewriting both the outer and inner loops. It is also notable that the original differential equation (the intent of the model) is completely absent as is the mathematical analysis which lead to the discrete approximation. All of these issues arise because of the conflation of the three conceptual aspects above.

Imperative programming languages such as C and Fortran closely match their execution to sequential (non-parallel) hardware, rather than a more abstract, declarative description of a computation. This makes it difficult to convert programs to new architectures (*e.g.*, heterogeneous, parallel systems) and thus to exploit their full performance benefits. Programs may be rewritten to target a particular architecture. For example, OpenMP extensions can be used for shared-memory parallelism, requiring only lightweight program annotation [DM98]. Other parallelisation strategies require more significant rewrites to the program, for example a GPU-based solution using CUDA primitives [Hal08] or a distributed-parallel solution using MPI [SO98]. However, all of these approaches require the original sequential version to be rewritten, leaving the program targeted to a particular hardware approach; a multi-platform, or *platform agnostic* approach is not provided. This deficiency in portability between hardware platforms is likely to worsen as hardware architectures evolve.

As an example, the heat flow program uses the `forall` abstraction (line 16) which provides an opportunity for data-parallel computation. However, the low-level nature of Fortran means execution of the outer loop, with the stateful double-buffering operation on line 15, has a fixed behaviour and cannot be (safely) parallelised.

2.4 Alternate approaches

Lower-level imperative languages such as C and Fortran remain popular, but there is increasing use of object-oriented languages such as Java, C++, and Python, and mathematics-oriented

```
1  import numpy
2  # parameters go here ... (akin to Figure~1, lines 1-9, modulo type declarations)
3
4  h = numpy.zeros([nx])
5  h[0] = 1
6  for t in range(0, nt):
7     h_old = numpy.copy(h)
8     h[1:-1] = h_old[1:-1] + r*(h_old[:-2] - 2*h_old[1:-1] + h_old[2:])
```

Figure 2: Python program with NumPy, solving an instance of the heat equation.

languages such as R, Matlab, and Mathematica. There are benefits to be found in all these new languages. However, at a fundamental level the programs they produce are very similar.

Figure 2 shows an equivalent solver to Figure 1, using Python with *NumPy* [ADH+01] which is increasingly popular for computational science [Oli07]. NumPy's benefits include a multi-dimensional array data type (abstracting machine implementations of arrays) and extensive library functions. The program is not vastly different to the Fortran; *topologically*, they are identical. Python's slicing syntax is combined with NumPy's operator overloading (of `+`/`-`, line 8)[1] to provide the inner loop which is similar in essence to Fortran's `forall`. However, this requires some thought about the slicing domains, which are more complex than relative offsets.

Object orientation does not improve the clarity of the model here. Instead, object-orientation simply provides a means for extending Python, *e.g.*, via operator overloading.

3 Towards programming language solutions

We propose that the problems of the previous section can be addressed by new programming constructs, targeted to computational science. We advocate for (1) high-level *abstractions* in programming, coupled with (2) *restrictions* on code and (3) lightweight verification techniques via *specifications*. Abstractions provided by programming languages support the development of complex software by hiding lower-level details and/or supporting code reuse. Restrictions to the expressivity of certain parts of a language support reasoning and optimisation, for example, `forall` statements are restricted to pure (side-effect free) expressions. Specifications aid the design process and can be used to automatically verify code when coupled with automatic tools.

In this section, we discuss potential approaches to these three aspects (abstraction, restriction, specification). We illustrate these with examples relating to our heat flow example, but imagine much wider applications to other kinds of model.

Functional programming We draw inspiration from functional and declarative programming approaches which have previously been argued as applicable to science (*e.g.*, [Hin09, Kar99]). Functional languages (such as F#, Haskell, ML, OCaml) more closely resemble mathematics than the execution approach of hardware (*cf.*, imperative languages). Functional languages therefore tend to more clearly express the higher-level concepts behind a program. This aids verification, understanding, and efficient execution across platforms.

Functional language abstractions are often coupled with restrictions to provide greater control over certain aspects of computation, most prominently over state and side effects. This allows programs to be treated more mathematically, aiding verification and optimisation via

[1] In normal Python, `x[1:n]+y[1:n]`, for example, concatenates array slices. With NumPy, this code instead computes the pointwise sum of the array slices.

equational reasoning [Bac78]. Such optimisations mean that functional languages can now out-perform traditional imperative languages, with less programming effort (*e.g.*, [MLPJ13]).

Functional languages also frequently provide verification support through advanced and flexible type systems, into which various program invariants and properties can be encoded and then automatically enforced at compile time. As an example of its flexibility, Haskell's type system can be used to encode the bounds-checking of relative-array indexing against the boundary (exterior) size of an array to guarantee code is free from *out-of-bounds errors*, without incurring runtime bounds-checking overhead [OM11]. These advanced type systems encode not just an abstract specification of *what* is computed (*e.g.*, a real number), but also *how* it is computed (*e.g.*, this computation uses a memory cell and accesses the $\texttt{i}^{th}$ element of array `x`). The information encoded in types can then be further leveraged for program optimisation.

Evolutionary approach Given the significant investment in current languages we propose an evolutionary approach to introducing new programming features in three stages:

1. *supporting existing models*: with language extensions and refactoring tools;
2. *extending modern language approaches*: leveraging the flexibility provided in existing languages (*e.g.*, overloading, object systems, type systems);
3. *creating the next generation of languages*: learning from current needs and practices.

Due to the evolutionary history of some languages (*e.g.*, Fortran, through many language standards), models developed and used over many decades are often *sedimentary*, containing layers of code using older features and idioms which are now deprecated and known to be a likely source of programmer error. Furthermore, these older layers do not exploit more recent abstractions.

Automatic refactoring tools provide a pathway to evolving and upgrading a code base to make better use of existing language features [OJ09, TM12]. For example, *Photran* is a Fortran refactoring tool for improving code clarity [Pho13]. We built a related automatic refactoring tool for Fortran (called *CamFort*) which eliminates uses of older Fortran features for manual memory and data management, replacing these with equivalent code in a modern style [OR13]. Examples include *common-block* elimination (replacing these with *modules*) and replacing manual data structuring idioms, using arrays as *records*, with the *derived data type* abstraction of Fortran 90. This improves code readability and often exposes potential sources of error. The static analysis techniques of CamFort are powerful enough to detect and apply many more transformations such as, replacing loops with `forall` (when the loops are of the correct form), which then benefits code portability to parallel architectures.

The rest of this section explores abstractions, specifications, and restrictions to improve computational science programming, and makes reference to supporting existing models as well as developing new languages and language extensions. We focus on static techniques to specification and restriction, *i.e.*, those that can be reasoned about by considering only the source code of the program. The alternative to this would be to fall back to dynamic checks (as in Python) which require reasoning about the execution as well as the source code. Although run-time techniques are easy to implement (consider assertion statements in many imperative/object-oriented languages) it is hard to assess whether sufficient constraints have been included to preserve the desired properties of the program. We note that there are various attempts to extend dynamically-typed languages such as Python with gradually-introduced static typing (see, *e.g.*, [LG11]) which may provide a pathway to static methods for such languages.

3.1 Specification systems

Specification systems are a lightweight, automatic verification technique. Specifications can be used to guide the development process (design-by-specification) or to verify existing models (where code is checked against a specification). We consider type systems as an example of a specification system, where type properties are specified explicitly in code or inferred, and programs violating the specification are rejected.

We discuss here dimension types (previously proposed in the literature), stencil specifications relating to our heat float example (new here), and a number of other type-related approaches in the literature that could be leveraged in computational science.

Dimension/units-of-measure typing Dimensional analysis is a simple method for checking basic consistency of a mathematical model where, for example, two quantities can only be added if they are of the same *dimension* (*e.g.*, both are length values) and *unit* (*e.g.*, both are values in metres). A *dimension type system* automates this analysis in programming, where numerical types can be assigned information on their dimension and/or unit [Ken94]. Anecdotally, we have observed that scientists sometimes include dimension/unit information in source code comments. Dimension types allows this information to be part of the language, giving them semantic meaning and power.

Consider the types of real-number addition and multiplication operations, which we write as $+, \times : (\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ (binary functions). With dimension types, real number types are annotated with a unit/dimension $\mathbb{R}^d$. The types of addition and multiplication are then:

$$+ : \forall d.(\mathbb{R}^d, \mathbb{R}^d) \to \mathbb{R}^d \qquad \times : \forall d_1, d_2.(\mathbb{R}^{d_1}, \mathbb{R}^{d_2}) \to \mathbb{R}^{d_1 d_2}$$

i.e., only quantities of the same unit can be added and multiplying values multiplies their units. The universal quantification $\forall$ here means these operations are *dimension polymorphic* (generic in their input dimensions). The benefits of such a system are highlighted by the well-known example of an uncaught unit mismatch leading to the destruction of the Mars Climate Orbiter [SMB+99]. Units-of-measure typing is now provided by F# [Ken08].

The benefits of dimension typing can be provided to existing models written in other languages via a pre-processor, extending the language. In this approach, existing code is augmented with comments, written with suitable lightweight syntax for dimension types, which are then checked by a pre-processor stage before compilation. For example, the following adds unit types to a fragment of the Fortran heat flow program:

```
! unit(s)   :: tmax
real, parameter    :: tmax = ...            ! end time
! unit(m)   :: xmax
real, parameter    :: xmax = ...            ! length of material
```

Since the unit types are comments they are ignored by the usual compilation tools.

Dimension typing for Fortran has been discussed for many years and there has been a recent ISO standard proposal to add this to the language standard [ISO13]. This ISO proposal follows Fortran's approach to typing, where every variable must have a declared unit type and every unit must itself be declared. Applying this approach to existing, large code cases therefore requires significant programmer effort. Instead, type inference can be used to infer the dimension types of unspecified parts of the program, and therefore reduce the number of required declarations.

We have prototyped a dimension typing system as part of the CamFort tool which resembles the ISO proposal but with polymorphic type inference and increased flexibility. This reduces programmer effort in applying the technique to existing models. We omit further details of the approach for brevity here.

Implementation specifications In Section 2.1, the discretization of the heat equation was performed by hand, making a number of choices. The solution uses the finite difference technique with the *Forward Time, Centered Space* (FTCS) scheme. This means that any error is *bounded above* by Δt and $(\Delta x)^2$. Furthermore, if $\mathtt{r} = \frac{\alpha \Delta t}{\Delta x^2} < \frac{1}{2}$ then the FTCS solution is known to be *stable* [Rec04]. This information is lost in the implementation of Figure 1. We propose that details of the discretization process should instead be part of the code to help verify and test the implementation, and to better communicate the solution approach and assumptions.

One approach would be to encode the *shape* of the array indexing pattern following from the FTCS scheme and the expansion depth of the Taylor series approximation. In our example, this leads to a centered (symmetrical) spatial pattern to a depth of one element in each direction. This could be encoded as a specification on the `forall` statement. For example:

```
! h : {symmetrical 1}
forall (x = 2:(nx - 1))  h(x) = h_old(x) + r*(h_old(x-1) - 2*h_old(x) + h_old(x+1))
```

i.e., relative to the current index `x`, `h` is accessed with a symmetrical pattern to depth 1. The program can then be automatically checked against the specification in a pre-processor stage to ensure that this approach was implemented correctly. The specification language should allow combinations of forward, backward, and centered (symmetrical) shapes in different dimensions.

Stencil specifications can be seen as a kind of computation type which provides information on *how* a computation is performed (in terms of data usage) and not just *what* is computed, as in traditional type systems.

Other typed approaches Other useful specifications for computational science might include value constraints, for example using *refinement types* [GF10] to specify the stability constraint of the FTCS scheme for the heat equations, *e.g.*, `real {< 0.5}, parameter :: r`.

More advanced type systems may be leveraged, for example *dependent types* which allow types to depend on values, elucidating the role of parameters in a computation and providing a richer description of computation [Hof97]. Dependent types have already been used in the area of climate impact research [IJ13a]. This work formalised measures of vulnerability to climate change using types which were then used to guarantee the correctness of programs implementing such measures. In computational science, such guarantees are rare, but dependent types can help scientists to write betters specifications and partial proofs of correctness leading to increasingly correct scientific computing [IJ13b].

By combining different kinds of type system (dependent, refinement, dimension, data access) future computational science languages can provide a rich set of automatic verification choices.

3.2 New abstractions and a language paradigm

Programming language abstractions benefit the reading, writing, understanding, verification, and high-performance execution of programs [Orc11]. This section considers abstractions for improving the communication of the scientific intent of a model, using our heat flow example.

Language features for general abstraction, such as higher-order functions coupled with parametric polymorphic types, can be used to build application-specific abstractions, for example, as in the approach of *algorithmic skeletons* for parallelism [Col89]. Application-specific abstractions are particularly effective when derived from data on the programming needs of practioners.

Previous studies have identified various core parallel programming patterns (many of which appear in computational science models) [ABD+08]. Such studies point the way towards useful new language abstractions for the computational science domain. For example, we previously developed a new notation for working with *structured grid* programs (such as finite difference

```
1  module heat {alpha : Real{m^2/s}}
2     model : PDE (X : Real{m}) * (T : Real{s}) -> (E : Real{j})
3     model h = d h T == alpha * (d (d h X) X)
4
5  instance heat(FTCS)
6     approx : {forward T 1, symmetrical X 1}
7     approx h [x, t] = let r = alpha * (delta(T) / delta(X)^2)
8                       in  h [x, t-1] + r * (h [x+1, t-1] - 2*(h [x, t-1]) + h [x-1, t-1])
9
10 instance heat(FTCS[Dirichlet[1]]) {x0 : Real{j}, xn : Real{j}, nx : Int, nt : Int}
11    delta(T)       = 0.05                          -- time step size
12    delta(X)       = 0.05                          -- space step size
13    extent(X)      = (0, nx)                       -- size of X (for some nx)
14    extent(T)      = (0, nt)                       -- size of T (for some nt)
15    boundary(T)[0] = array (0 -> x0, * -> xn)      -- set up initial array
16    boundary(T)[nt] = solve approx boundary(T)[0]  -- solve to get upper extent in T
17
18 h' = heat(FTCS[Dirichlet[1]]) {alpha = 0.1, x0 = 1.0, xn = 0.0, nx = 40, nt = 50}
```

Figure 3: A mockup language, separating model specification from solution implementation.

models) that reduces verification problems by replacing indexing offsets with a fixed "grid pattern" notation, resembling diagrams of stencil access patterns [OM11, OBM10]. This new language construct, coupled with algorithmic-skeleton style libraries, provides various automated verification and parallelisation opportunities.

Abstractions for separating concerns Despite advanced abstractions in functional languages, functional programs still often suffer from the problem of conflating concerns in the context of computational science. We thus propose here a *multi-level language* approach where higher-level abstract specifications of a model can be described separately to the lower-level numerical approximation and prediction process which refine the model.

For example, a multi-level language might allow the original differential heat equation to be directly encoded in the program as the abstract model (thus defining an essential property of the heat function) along with a solution strategy which gives a concrete definition (of the same type). This has several benefits. Firstly, the high-level language can communicate the scientific intent, unencumbered by implementation assumptions and possible bugs in low-level solution code. Secondly, validation of the model and verification of the implementation will be apportioned to the higher- and lower-level parts respectively. Thirdly, a model can be kept abstract but instantiated with a number of different solution methods which may have their own low-level implementation, tuned to different hardware platforms and architecture (*e.g.*, exploiting parallelism).

Figure 3 uses a hypothetical multi-level language for the heat flow example. The code is split into an abstract model (lines 1-3), FTCS approximation approach (lines 5-8), and instance of the approximation (lines 10-16). The example includes dimension types (lines 1-2,10) and stencil specifications (line 6). Lines 1-3 give the abstract specification of the heat equation, where `d` is a higher-order function which captures a partial derivative and is parameterised by a function and a dimension (*e.g.*, `T` or `X`). Lines 5-8 describe our FTCS approximation but does not specialise it to the problem size, resolution, or boundary conditions. Lines 10-16 then give an instance of the approximating solution, defining problem size and boundaries, which is instantiated with values on line 18. On lines 1 and 10, parameters of the model and solution respectively are declared, inside braces { and }, along with their unit types.

The abstract model specification in this approach (lines 1-3) is more than just a comment. It specifies solution behaviour and thus any approximate solutions can be *backchecked* at runtime against this model (see, *e.g.*, backchecking in [RB06]). For example, the FTCS predictions can be plugged back into the abstract model, where standard numerical approximations are automatically computed for differentials on each side of the equation. These can then be checked against a specification of expected error growth, or reported back to the programmer for analysis (for example, reporting on the absolute error between the two sides of the specification's equality on line 3). Further, the abstractions provided here allow for other automated testing approaches. For example, the parameter space for α can be explored, using a QuickCheck-style approach [CH00] to test the stability and robustness of the solution approach.

There are a number of useful language restrictions used here. Firstly, expressions between brackets, [and], must be *affine* expressions of indices, thus restricting indexing to a decidable fragment of arithmetic. This provides decidable information to the compiler about the access pattern which could be used for verification, or to guarantee layout optimisation and parallelisation strategies (*e.g.*, domain decompositions with minimal *halos*). Another restriction might be that functions using state must declare so in a specification, which must then be propagated to the specifications of all functions that use it (*cf.*, the *monadic* typing discipline [Wad95]). In the example, the functions are free from state, and thus the compiler knows that the solution method is data parallel (with only a backward, temporal dependency following from line 8).

We have experimented with this new programming approach in Haskell, where we have developed a prototype for specifying partial-differential equation (PDE) models and automatically checking approximations against these model specifications.[2] The prototype does not include all features discussed here but supports the main approach of describing an abstract model in code, separate to an implementation. Library functions then provide automatic testing of the suitability of an approximation from the PDE specification. Appendix A shows an example using our prototype that resembles Figure 3. Our prototype validates the hypothesis that this is a plausible programming approach.

Related to this approach of expressing models in code, the FEniCS project provides a programming system for the automatic solution of partial-differential equations which embeds differential equations in code (C++ and Python, although the C++ approach is more powerful) [LMW12]. FEniCS provides its own system for solving PDEs and does not provide a mechanism for coupling manual solutions with the PDE specifications.

The language approach described here, and provided by our prototype, supports reproducibility by providing programming constructs for clearly expressing an abstract model in code which can then be shared separately to any approximations, in a modular fashion. Furthermore, the approach supports verification by providing automated testing of approximations against the abstract model. Future programming systems for computational science should provide similar abstractions for separating concerns in different kinds of model (not just PDEs), and thus aiding understanding, reproducibility, and verification.

4 Conclusion

Computer modelling has benefited the scientific method by providing tools to develop increasingly complex theories. However, at the same time this has substantially increased the complexity of making predictions from a hypothesis and reproducing results. We discussed how programming language concepts can be applied to the computational science domain to pro-

[2]The prototype is available with documentation at `https://github.com/dorchard/pde-specs`.

vide more effective languages. The intention is to reduce the burden on scientists by evolving existing tools and languages in new directions. We have begun this evolutionary process with CamFort, which can apply code transformations to Fortran code-bases and type-check unit-type annotations (embedded via comments).

We believe there are benefits available from moving towards a multi-language approach. Initially this could take the form of attaching relevant specifications to existing code (such as dimension typing or stencil specifications as demonstrated). We outlined briefly a new programming language approach that more thoroughly separates the intent of a model from its implementation. The novel approaches discussed here were however specialised to just one kind of model: partial-differential equations, although stencil specifications are useful for other kinds of model with regular, data-access patterns on arrays (such as cellular automata).

The general idea of separating concerns via language abstractions is however widely applicable to many different kinds of modelling. What is needed now is increased collaboration between computational scientists and programming language researchers such that the state-of-the-art in computational science programming can progress to cover a wide variety of modelling approaches. There is a distinct lack of quantitative data on the programming patterns prevalent in computational science models, which should be remedied. Furthermore, to aid adoption, there is a definite need for user-focussed research, with usability analyses of any new programming approaches and serious efforts to integrate with the existing working practices of scientists.

There are many opportunities and there is much work to be done. The time is now to build new languages and tools to support the next generation of scientific research.

Acknowledgments Thanks are due to Oliver Chick, Michael Fischer, Cezar Ionescu, and Alan Mycroft for comments and discussion, Oleg Oshmyan for his work on units-of-measure typing for Fortran in CamFort, and Andy Hopper for his support. This research was supported by a Google Focused Research Award.

References

[ABD+08] Krste Asanovic, Ras Bodik, Demmel, et al. The Parallel Computing Laboratory at U.C. Berkeley: A research agenda based on the Berkeley view. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008.

[ADH+01] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant. Numerical Python. Technical Report UCRL-MA-128569, 2001. Lawrence Livermore National Laboratory.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[CH00] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[Col89] M.I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman, 1989.

[CRR+06] Geoffrey Chang, Christopher B. Roth, Christopher L. Reyes, Owen Pornillos, Yen-Ju Chen, and Andy P. Chen. Retraction. *Science*, 314(5807):1875, 2006.

[DDD91] C. Dawson, Q. Du, and T. Dupont. A finite difference domain decomposition algorithm for numerical solution of the heat equation. *Mathematics of Computation*, 57(195), 1991.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[Dru09] Chris Drummond. Replicability is not reproducibility: nor is it good science. In *Evaluation Methods for Machine Learning Workshop at the 26th ICML, Montreal, Canada*, 2009.

[GDN97] M. Griebel, T. Dornsheifer, and T. Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction*, volume 3. Society for Industrial Mathematics, 1997.

[GF10] Andrew D Gordon and Cédric Fournet. Principles and Applications of Refinement Types. *Logics and Languages for Reliability and Security*, 25:73–104, 2010.

[Hal08] Tom Halfhill. Parallel Processing with CUDA. *Microprocessor Report*, Januray 2008.

[Hin09] K. Hinsen. The promises of functional programming. *Computing in Science & Engineering*, 11(4):86–90, 2009.

[Hof97] M. Hofmann. Syntax and semantics of dependent types. *Semantics and logics of computation*, 14:79, 1997.

[IJ13a] Cezar Ionescu and Patrik Jansson. Dependently-Typed Programming in Scientific Computing - Examples from Economic Modelling. In *IFL (Implementation and Application of Functional Languages) 2012*, volume 8241 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2013.

[IJ13b] Cezar Ionescu and Patrik Jansson. Testing versus proving in climate impact research. In *Proceedings of the 18th Workshop Types for Proofs and Programs (TYPES'11)*, volume 19, pages 41–54, 2013.

[ISO13] ISO/IEC JTC1/SC22/WG5, Units of measure for numerical quantities, April 2013. N1696, `ftp://ftp.nag.co.uk/sc22wg5/N1951-N2000/N1969.pdf`.

[Kar99] L. Karczmarczuk. Scientific computation and functional programming. *Computing in Science & Engineering*, 1(3):64–72, 1999.

[Ken94] Andrew Kennedy. Dimension types. In *European Sympsium on Programming Languages and Systems (ESOP)*, pages 348–362. Springer Berlin Heidelberg, 1994.

[Ken08] Andrew Kennedy. Types for units-of-measure in F#: invited talk. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 1–2. ACM, 2008.

[KKZ07] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 7–1. ACM, 2007.

[LG11] Jukka Lehtosalo and David J Greaves. Language with a Pluggable Type System and Optional Runtime Monitoring of Type Errors. In *Proceedings of International Workshop on Scripts to Programs (STOP)*, 2011.

[LMW12] A. Logg, K.A. Mardal, and G. Wells. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Lecture Notes in Computational Science and Engineering. Springer, 2012.

[Mer10] Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature*, 467(7317):775–777, 2010.

[MLPJ13] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. Exploiting Vector Instructions with Generalized Stream Fusion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 37–48, 2013.

[OBM10] Dominic Orchard, Max Bolingbroke, and Alan Mycroft. Ypnos: declarative, parallel structured grid programming. In *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 15–24, New York, USA, 2010. ACM.

[OJ09] Jeffrey L Overbey and Ralph E Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. In *ACM SIGPLAN Notices*, volume 44, pages 493–502, 2009.

[Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[OM11] Dominic Orchard and Alan Mycroft. Efficient and Correct Stencil Computation via Pat-

tern Matching and Static Typing. *Electronic Proceedings in Theoretical Computer Science*, 66:68–92, 2011. arXiv:1109.0777.

[OR10] W.L. Oberkampf and C.J. Roy. *Verification and validation in scientific computing*. Cambridge University Press, 2010.

[OR13] Dominic Orchard and Andrew Rice. Upgrading Fortran Source Code Using Automatic Refactoring. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, WRT '13, pages 29–32, New York, NY, USA, 2013. ACM.

[Orc11] Dominic Orchard. The four Rs of programming language design. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 157–162. ACM, 2011.

[Pho13] Photran – An Integrated Development Environment and Refactoring Tool for Fortran, Retrieved July 2013. `http://www.eclipse.org/photran/`.

[PV05] D.E. Post and L.G. Votta. Computational science demands a new paradigm. *Physics today*, 58(1):35–41, 2005.

[RB06] Andrew C Rice and Alastair R Beresford. Dependability and accountability for context-aware middleware systems. In *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on*, pages 5–pp. IEEE, 2006.

[Rec04] G.W. Recktenwald. Finite-difference approximations to the heat equation. *Class Notes*, 2004. `http://www.f.kth.se/~jjalap/numme/FDheat.pdf`.

[She05] T. Sheard. Putting Curry-Howard to work. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85. ACM, 2005.

[SMB+99] Arthur G Stephenson, Daniel R Mulville, Frank H Bauer, Greg A Dukeman, Peter Norvig, LS LaPiana, PJ Rutledge, D Folta, and R Sackheim. Mars Climate Orbiter Mishap Investigation Board Phase I Report, 44 pp. *NASA, Washington, DC*, 1999.

[SO98] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[SV07] M. Sulzmann and R. Voicu. Language-based program verification via expressive types. *Electronic Notes in Theoretical Computer Science*, 174(7):129–147, 2007.

[TM12] Fernando G Tinetti and Mariano Méndez. Fortran legacy software: source code update and possible parallelisation issues. In *ACM SIGPLAN Fortran Forum*, volume 31, pages 5–22, 2012.

[Var10] Moshe Y. Vardi. Science has only two legs. *Commun. ACM*, 53(9):5–5, September 2010.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

A Example of PDE models in Haskell

Available at `http://github.com/dorchard/pde-specs/` in the file `ICCS-example.hs`.

Line 5 specifies the heat flow PDE, `spec`. Lines 8-13 describe the FTCS approximation as a recurrence relation, `approx`, with Dirichlet boundaries. Line 18 uses a memoizing combinator `arrayMemoFix` to provide a high-performance version of the recurrence, `approxFast`. Note the type signature on lines 16-17 show that the approximation has a discrete domain (a pair of integers) and requires *implicit* parameters, `?nx`, `?nt`, `?dt`, `?dx`, and `?alpha`. Lines 20-26 instantiate the approximation (providing values for the implicit parameters) and then runs `verifyModel` (line 26) with specification `spec` and approximation `approxFast`. This computes the absolute error between the left- and right-hand sides of the PDE `spec` applied to the `approxFast` solution, using the Euler method for its numerical approximation of the differentials.

```
{-# LANGUAGE ImplicitParams #-}
import PDESpec

-- Specification of heat equation PDE
spec h = (d h T) === (constant ?alpha * d2 h X) `withDomain` (X :. T :. Nil)

-- Implementation using a recurrence relation
approx h' (x, t)
    | x == 0    = 1
    | x == ?nx  = 0
    | t == 0    = 0
    | otherwise = h' (x, t-1) + r * (h' (x+1, t-1) - 2 * h' (x, t-1) + h' (x-1, t-1))
                  where r = ?alpha * (?dt / (?dx * ?dx))

-- Create a fast version using a memoizing combinator arrayMemoFix
approxFast :: (?nx :: Int, ?nt :: Int, ?dx :: Float, ?dt :: Float, ?alpha :: Float)
              => (Int, Int) -> Float
approxFast = arrayMemoFix ((0, 0), (?nx, ?nt)) approx

-- Instantiate approximation; verify by calculating abs error of 'approxFast' from the spec
experiment = let ?dx = 0.05
                 ?dt = 0.05
                 ?nx = 100
                 ?nt = 100
                 ?alpha = 0.006
             in verifyModel Euler spec approxFast
```