Published paper

Duke, D.J., Borgo, R., Runciman, C. and Wallace, M. (2009) *Huge data but small programs: visualization design via multiple embedded DSLs.* In: Practical Aspects of Declarative Languages : 11th International Symposium, PADL 2009, Savannah, GA, USA, January 19-20, 2009, Proceedings. Lecture Notes in Computer Science (5418). Springer , pp. 31-45. ISBN 978-3-540-92994-9

http://dx.doi.org/10.1007/978-3-540-92995-6_3

Huge Data but Small Programs: Visualization Design via Multiple Embedded DSLs

D.J. Duke[1], R. Borgo[1], C. Runciman[2], and M. Wallace[2]

[1] School of Computing, Uni. of Leeds, UK,
{djd,rborgo}@comp.leeds.ac.uk
[2] Dept. of Computer Science, Uni. of York, UK,
{colin,malcolm}@cs.york.ac.uk

Abstract. Although applications of functional programming are diverse, most examples deal with modest amounts of data – no more than a few megabytes. This paper describes how Haskell has been used to address a challenging visualization problem, involving 200 time steps from an astrophysics simulation of early star formation. Each individual step involves nearly 40 million samples, and uncompressed the complete dataset is nearly a terabyte.

Our solution makes novel and extensive use of *domain specific languages* to specify data resources, rendering abstractions, and most significantly, the desired visualization. The result is a powerful framework for multi-field visualization, including the use of animation to explore the evolution of data over time. This approach represents a significant departure from standard practices in visualization, and has applications well beyond the original problem. That our solution consists of less than 4.5K lines of code is itself a notable result. This paper motivates and describes the overall architecture of our solution, and technical features of the DSLs that are used in place of the traditional visualization pipeline. We conclude with thoughts on how functional technologies may find even broader use within other branches of visualization.

1 Introduction

Drawings, diagrams and graphs have a long history of use within scientific discovery, e.g. Snow's map correlating cholera cases with water pump location in London, 1854. Use of computer graphics for visualizing data is usually traced to an influential report produced for the US NSF, and published in 1987 [1]. Data from instruments and supercomputer simulation was accumulating faster than it could be interpreted, and the report called for new methods to process these 'firehoses'. Visualization became established as a new research field within computing, and foundational work on data models, processing paradigms and depiction techniques for large-scale data led to rapid progress [2, 3]. Much of this work concentrated on *scientific* visualization, where the data are located within some physical space. Data that has no 'natural' spatial component, for example metabolic networks, web sites, market trends, etc., is addressed by *information* visualization. Although they have similar aims, the relationship between these two branches of visualization has been the subject of much debate [4]. Our concern is with scientific visualization, but we will return to consider information visualization in the conclusion.

The remainder of the paper reports on how Haskell has been used to address a major visualization design challenge, the 2008 IEEE Visualization Design Contest [5]. Section 2 introduces the contest and explains its importance and relevance to practical applications of scientific visualization. Our solution [6] utilises a two-stage pipeline, separating the management of datasets from the synthesis of pictures. The architecture is described in Section 3, with data management and picture synthesis forming sections 4 and 5. Section 6 sets out an evaluation of our work. We contrast our approach to the contest with entries from previous years, and reflect on the design decisions that were made. In the conclusion, Section 7, we pay particular attention to our use of domain-specific languages, and their further potential within visualization.

2 The IEEE Visualization Design Contest

Since its inception in 1990, IEEE Visualization has been the leading forum for research in the field. Along with InfoVis (information visualization) and VAST (visual analytics), the conference now forms one of the three strands within 'VisWeek'. In 2004, the conferences instituted a visualization contest, designed "... to foster comparison of novel and established techniques, provide benchmarks for the community, and to create an exciting venue for discussion ...".

The logistical difficulties presented by the contest can be appreciated from an outline of the 2008 edition [5]. The dataset comprises 200 timesteps from an astrophysics simulation, modelling interaction between a radiation ionisation front and primordial gas within a $0.6 \times 0.25 \times 0.25$-parsec volume of space (sampled as a regular $600 \times 248 \times 248$-point grid). Understanding this interaction would provide new insight into structure formation in the early universe, and the contest itself sought answers to six specific questions relating to these interactions. At each point in the space, the simulation tracks ten scalars and one vector, with the scalars recording temperature and density of the gas, and the relative densities of 8 chemical species. Data are stored using a 10-character ascii representation of fixed-precision format numbers; uncompressed, the total size of the dataset would be 960GB. Fortunately, as we will explain, it is possible to work with rather smaller subsets.

A further indication of the difficulty can be seen in the 6-month contest timeframe, and the number of entries received. The visualization conferences regularly attract around 750 delegates, but for the previous contest in 2006[3] (an earthquake simulation), only 6 entries were received, and of these only three, including the winner, received an explicit mention. Contrast this with the 72-hour ICFP programming challenge, which draws around 350 teams relative to

[3] For logistical reasons, there was no contest in 2007.

a core conference community of 250. Tackling the visualization contest requires access to domain expertise, robust and scalable software, and significant time to explore the problem and solution space. Past entries have used mature off-the-shelf systems, either commercial products (including the open-source VTK), or the output of long-running research initiatives.

In a series of papers [7–10] we have explored the use of functional language technologies (and specifically Haskell) to reconstruct visualization techniques, taking advantage of lazy evaluation to implement streaming of data, and the expressive type system to create new kinds of generic abstraction. This work provides a necessary foundation for our solution. However, it was not in itself sufficient. Central to the 2008 design contest is the problem of time-varying multi-field data, a challenge in many visualization applications. Although our previous implementations supported a combination of techniques, for the most part they only supported visualization of a single field within a single timestep.

3 Architecture of a Solution

Before designing a solution, we need first to unpack the problem. Visualization is used in three ways: to present known data, to confirm a known hypothesis, or to discover what might be present within unseen data. The six contest questions fall into the latter two categories. Five ask about interactions between specific fields. For example, here is question two:

> "Over 100 chemical reactions occur in primordial H and He (many of which are driven by radiation in the I-front) but what most interests those studying first structure formation in the universe is H_2. It allowed primeval gas clouds to collapse and form the first stars before galaxies later coalesced. Where is H_2 most prevalent in the simulation?" [5]

Although this question only mentions one field (H_2) explicitly, the answer has be framed in terms of the relationship between H_2 concentrations and other features, e.g. the hottest regions, and the advancing I-front. This requires multiple fields. The final question is more open-ended and invites wholesale exploration:

> "Question 5 posed a very specific hypothesis about the cause of turbulence. The broader question of interest, and the one for which visualization offers the most promise of displaying something unexpected, is 'What is causing the turbulence?' Can you do an open-ended visualization of all variables to try and help answer this question? This is the 'seeing the unexpected' question that will hopefully provide new hypotheses." [5]

Putting aside the temporal element for now, there are two general strategies for dealing with multi-field data. (1), combine a number of standard techniques; for example, extracting an isosurface from one field and colouring it by probing into a second field, or by using multiple cutting planes. Or (2), use a visual technique designed specifically to expose relationships between fields. *Scatterplots* can be used for two or three fields, while *parallel coordinates* generalise to higher dimensions [11], but in both cases it is difficult to see correlation with 3D spatial locations, and correlation with spatial features (e.g. the shockwave) mentioned in the contest questions. These needs could be addressed by *brushing* and other forms of interaction, but we took an early decision to focus our work on the first strategy, combining standard techniques within the physical space of the simulation.

For dealing with time, there are again two general strategies; either (1) represent it explicitly as a spatial dimension, for example plotting a graph with time as one axis, or (2) represent it implicitly, by using animation. Following a meeting with astrophysicists to obtain a better understanding of the problem, we were encouraged to explore animation. As we will see, our solution actually creates interesting possibilities for combining time and space within one representation.

Our first design decision was to split our solution into two stages:

Stage I: Data Management – conversion of datasets into a more compact binary representation, support for fixed-precision calculation, selection of fields, slicing, and downsampling.

Stage II: Picture Synthesis – specification of picture parameters, selection of files, synthesis and rendering of geometry, and interaction.

These stages were loosely coupled, driven by separate executables, and linked through the filesystem. Figure 1 shows this overall architecture, and in the next two sections we will explore the design of each stage in detail.

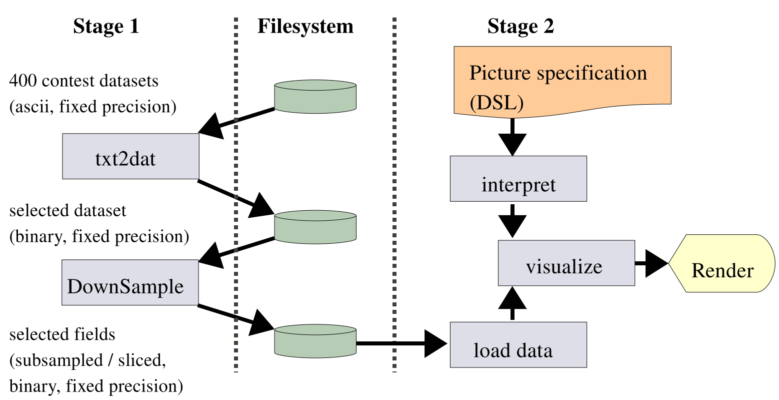

Fig. 1. System architecture.

The architecture maps onto the remainder of the paper as follows: Section 4 is concerned with the left-hand side of the diagram, and Section 5 with the right. Section 5.1 describes the 'Visualize' box, 5.2 the 'Render' box, and 5.3 unpacks the 'Picture specification DSL'.

4 Stage I: Data Management

The contest data consists of 400 primary files, 200 holding the scalar field values for each timestep, and a further 200 carrying the vector (velocity) data. Within a scalar file, the value for each of the 10 fields is given for the first point, then the 10 values for the second point, and so on. Consequently the entire file must be traversed, even if only one or two fields are of interest. We decided to define our own storage model for this data, and at the same time to convert the ASCII encoding into a more compact binary form.

4.1 Fixed-Precision Values

Numeric computation in visualization and computer graphics often uses the 32 or 64-bit IEEE floating point representation, and it would have been straightforward to convert the fixed-precision representation into this form. However, as part of the analysis we would need to carry out derivation of new fields from the existing data, for example computing the turbulence of the flow as the magnitude of the velocity field curl. The numerical ranges for some fields are large, and concerned about loss of precision we decided to work as much as possible using our own fixed-precision representation. Each value was represented in mantissa-exponent format, with 15 bits for each value (plus a sign bit). Internally, this format was stored using a Haskell constructor with two 16-bit integer components, while externally values could be stored as 4 bytes in a binary file.

Having adopted this representation, we then provided support for computations by a writing a small fixed-precision arithmetic library. The first step was to define the operations within Haskell. Defining the new representation as an instance of the *Num* type class provided overloaded arithmetic operators, but more importantly we utilised SmallCheck [12] to test expected properties of the system, for example commutativity:

$$\begin{array}{l} \mathit{prop\_plusCommutes} :: \mathit{FixedPrecision} \rightarrow \mathit{FixedPrecision} \rightarrow \mathit{Bool} \\ \mathit{prop\_plusCommutes}\ x\ y = x + y \equiv y + x \end{array}$$

This was invaluable in quickly teasing out a number of bugs. Just as importantly, having established confidence in the Haskell 'specification', we were able to use it as a reference model for implementing the fixed precision library within C. Functions in the C implementation were exposed to Haskell via the FFI, and equivalence between C and Haskell representations was tested via commuting-diagram properties, e.g.

$$\begin{array}{l} \mathit{prop\_times} :: \mathit{FixedPrecision} \rightarrow \mathit{FixedPrecision} \rightarrow \mathit{Bool} \\ \mathit{prop\_times}\ x\ y = (x * y) \equiv \mathit{fromCFP}\ (\mathit{toCFP}\ x * \mathit{toCFP}\ y) \end{array}$$

4.2 Downsampling

In transforming the representation, we have not addressed the resolution or bounds of the data. There are good reasons for *not* working directly from the full $600 \times 248 \times 248$-point grid at each timestep:

– A standard strategy in visualization is to first gain an overview of the data, and then descend into lower levels of detail, saving unnecessary computation.
– Our volume renderer has a very simple implementation, but one based on nested lists, and could not render the volume at full resolution.
– Our astrophysics colleagues had suggested that for a number of the contest tasks, 2D slices might provide a more useful view.

We will return to *why* 2D slices are useful in the next section, but the conclusion from these three points is that we needed a flexible mechanism for extracting subsets of the data, both by downsampling, and/or by restricting the range of one or more dimensions. Our implementation consisted of three components:

– a regular *naming scheme* for resources (files) that encodes information about the spatial bounds, sampling, and fields;
– a high-level *planner* that, given the specification of a required resource, computes the cheapest plan for generating that resource from the available files; and
– a *worker* program that implements a given plan.

Three examples of the resource naming conventions are:

```
x0-599y0-247z0-247t10.DGHH+HeHe+He++H-H2H2+.dat
x0-4-599y0-4-247z0-4-247t100.H2.dat
x0-599y0-247z124t60.G.dat
```

The first example specifies a full-resolution sampling of the entire grid, at time step 10, containing each of the 10 scalar attributes (D, G, H, H+, etc). In the second specification, the grid at time 100 has been downsampled, with every 4th sample selected in each spatial dimension, and only the $H2$ scalar component selected. The final example specifies a 2D slice at time step 60 corresponding to the plane $z = 124$, with full resolution along the remaining two axis, and carrying the G field.

The *planner*, implemented in Haskell, takes a resource specification as parameter, and then inspects the available files, deciding the cheapest method for generating the resource. Selection is implemented by defining a partial order over data files. This is an inclusion relation defined in terms of data-files' bounds (spatial and temporal), granularity (spatial and temporal) and the set of fields present. After selecting the least dominator under the ordering, the planner invokes a *worker*. The worker, implemented in C for performance reasons, converts the plan into a tight set of nested for-loops that traverse the input and generate the output resource. It takes the worker around two minutes to downsample/slice from the largest resource file (1.48Gb), so planning to start from the least dominator can be a significant time win. In the case of *derived* fields, part of the worker traversal involves per-point numeric computation over selected samples from the input.

5 Stage II: Picture Synthesis

Before the announcement of the design contest, we had already implemented a modest library of *3D* visualization techniques, specifically

- isosurface extraction;
- hedgehog rendering of a vector field;
- probing; and
- pseudo-volume rendering.

Experience gained in implementing these algorithms is reported in [10]. For addressing the contest tasks, three further techniques were implemented:

- slice visualization;
- 2D contouring; and
- 3D scatterplot.

Building on the Stage I work, we were easily able to adapt our infrastructure to process contest datasets, obtaining initial results such as the volume rendering of gas density, and isosurfaces of gas temperature, shown in Figure 2.

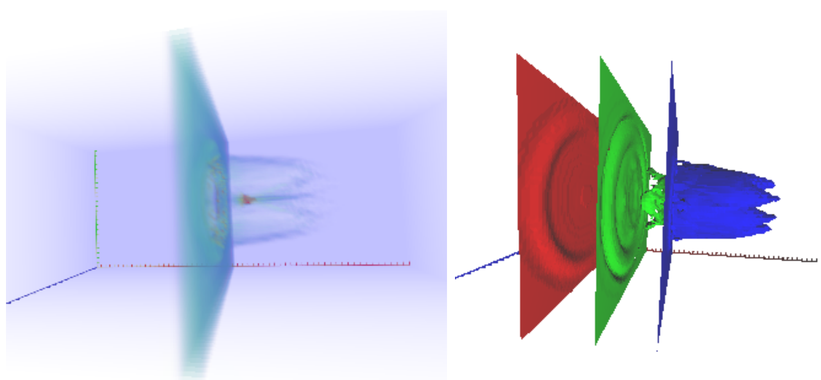

Fig. 2. Left: gas density as a volume rendering. Right: isosurfaces for gas temperature at 2.5K (blue), 16K (green) and 20K Kelvin (red). Both pictures are generated from time step 60, downsampled to a $150 \times 62 \times 62$ grid.

This figure highlights both the power of visualization to present data, and the limitations of standard 3D techniques for this particular challenge. The aim is to explore correlations between multiple fields. Superimposing 3D representations, even where they are known to be disjoint, creates problems of occlusion. This problem is avoided in Section 5.1 by utilising 2D techniques.

Although we could combine techniques, synthesising a compound image required an ad-hoc and complex set of command-line arguments to the viewer program. We were also limited to static views of a single timestep. So we reorganised our software around a domain-specific language (DSL) for multi-field visualization. This DSL will be described in Section 5.3, after Section 5.2 has introduced the rendering layer that mediates between visualization techniques and low-level graphical IO.

5.1 Contours and Slices

Isosurfaces are a 3D generalisation of an older method for depicting scalar fields, the contour plot. Contour plots have the advantage that nesting of contours can be easily seen and interpreted. Contouring a field at regular intervals also highlights areas of high gradient, a feature that we found useful in addressing one of the contest questions. Similarly, a 2D slice through a dataset can also be rendered directly, by using a transfer function to associate a colour with each point, and then smooth-shading the resulting mesh. Figure 3 shows the same datasets as Figure 2, this time using slicing and contouring on a single plane. We found that these images are more useful in revealing details of the underlying field. In particular the contour plot reveals a region of hot gas embedded within the shell of the shockwave. As we shall see, these representations are also more amenable to composition.

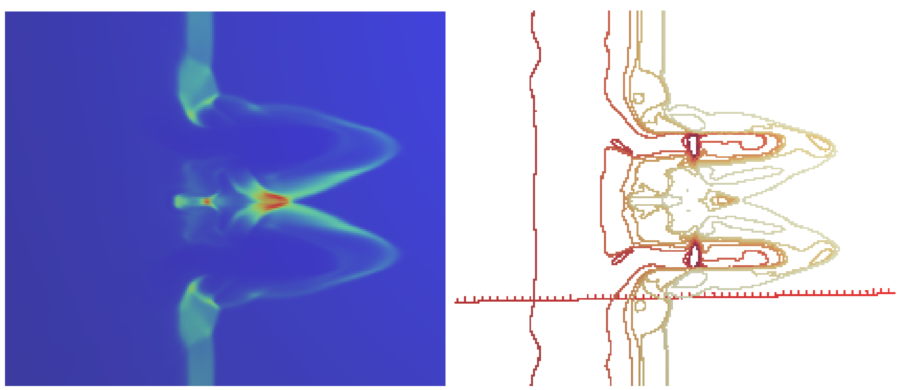

Fig. 3. Left: gas density as a slice. Right: contour lines for gas temperature, range 2K, 3K . . . 21K Kelvin. Both pictures again from time step 60, now at full resolution within the plane $z = 124$.

The *implementation* of contouring provides a compelling example of the value of abstraction, and Haskell's type class system. Following our initial work on the 'marching cubes' algorithm [9], we generalised our dataset representation and implementation of the algorithm. The signature of our isosurfacing algorithm now consists entirely of type variables and constraints:

$$\begin{aligned}&\mathit{isosurface} :: (\mathit{Interp}\ a, \mathit{InvInterp}\ a, \mathit{Interp}\ g, \mathit{Cell}\ c\ v, \mathit{Enum}\ v) \Rightarrow \\ &\quad a \rightarrow \mathit{Dataset}\ c\ v\ a \rightarrow \mathit{Dataset}\ c\ v\ g \rightarrow [[g]]\end{aligned}$$

$$isosurface\ th\ samples\ geom$$
$$= zipWith\ (surf\_cell\ th)\ (stream\ samples)\ (stream\ geom)$$

It requires three parameters: a threshold to be extracted, a stream of sample values, and a stream of the geometric locations at which the samples were obtained. The two streams are built from topological cells defining local neighbourhoods within the grid. A cell in turn is simply a type class that describes the capability to select a vertex, and a case table that maps a *marking*, indicating which vertices of the cell are above a threshold, to the list of edges that are intersected by the surface. It took us less than one hour to implement *2D* contouring as an instance of this generalised algorithm. We had only to:

1. define a data constructor for 2D (square) cells;
2. implement the two *Cell* methods - the case table consisting of just 16 lines;
3. implement a function to turn a stream of values (samples or geometry) into a stream of squares, a simpler instance of the technique described in [9]; and
4. wrap the output of the "isosurfacer" with the appropriate geometry for rendering at a set of line segments.

5.2 Rendering and Interaction

The output of a single visualization algorithm such as isosurfacing, contouring, or volume rendering, is a bag of primitives: coloured line segments, triangles, and surface normals. These must then be rendered to a display, in some fasion that allows for interactive exploration, e.g. rotation, translation and zooming of the "camera". Ultimately, the visualization front-end is implemented using the HOpenGL library that we have found to provide an excellent interface to OpenGL and GLUT. However, rendering and event handling in OpenGL are handled through callbacks, which represent an unfortunately low-level intrusion into the functional environment of our visualization system. To mitigate this, we have implemented an intermediate layer, in the form of a *scene-graph* [13] abstraction supporting a purely functional layer of event handling. This provides a DSL for graphics, and serves as the target language into which the picture DSL, described in the next section, is compiled:

```
type HsHandler a = Maybe (Event -> a -> a)
type HsMovie     = (Bool, [HsScene], [HsScene])

data HsScene
  = Camera     (HsHandler HsView)        HsView HsScene
  | Geometry   (HsHandler [HsGeom])      PrimitiveMode [HsGeom]
  | Transform  (HsHandler HsTransform)   HsTransform
  | Group      (HsHandler [HsScene])     [HsScene]
  | Compiled   HsCompiledHandler         Extent DisplayList
  | Switch                               HsScene HsScene HsScene
  | Imposter                             HsScene HsScene
  | Animate    (HsHandler HsMovie)       HsMovie
  | Special                              (IO ())
```

Nodes in the tree include: scene geometry, transformations, groups of subtrees, compiled scenes (OpenGL display lists), and animations. Each animation is stored as a pair of lists along with a 'playing' flag. The lists hold the frames yet to be played, and the frames that have been played. The *Animate* event handler can be instantiated with a basic movie player supporting playback, pausing, and stepping through individual frames. Lazy evaluation means that one frame can be on the display while the next frame is still being generated.

In response to OpenGL's callback architecture, the rendering module uses a global IORef to store the root of the scene. Most node types include an *event handler*, a pure function over the node's substructure. When an OpenGL callback is invoked, for example due to a mouse or timer event, the scene graph is traversed. At each node with a handler, a new node is generated by evaluating the handler with the event and node structure as parameters. After traversal of the tree, the new root node is written back to the IORef. Although this solution hides some of the non-functional features of OpenGL's architecture, there is clearly room for further improvement. One possible direction is work on functional reactive programming; the Yampa library has for example been used to create interactive graphics applications [14], though it is unclear how well this would interface with the structured approach to rendering adopted here.

5.3 The Picture DSL

Slicing and contouring yielded simpler views of a single timeframe, but our greatest win came from creating compound images and animations that exposed the relationship between fields over time. To achieve this, we wrote a small DSL of pictures that provides a task-oriented vocabulary, mediating between the rendering and data-management languages. A *picture* is either the output from one of our visualization techniques, or a compound of simpler pictures:

$$
\begin{array}{lll}
\mathbf{data}\ \mathit{Picture} = & \mathit{Contour} & \mathit{Colour}\ (\mathit{Range}\ \mathit{Float})\ \mathit{DataExpr} \\
\quad\mid & \mathit{Surface} & \mathit{Colour}\ (\mathit{Range}\ \mathit{Float})\ \mathit{DataExpr} \\
\quad\mid & \mathit{Volume} & \mathit{Colour}\ \mathit{DataExpr} \\
\quad\mid & \mathit{Slice} & \mathit{Colour}\ \mathit{DataExpr} \\
\quad\mid & \mathit{Scatter} & \mathit{DataExpr}\ \mathit{DataExpr}\ \mathit{DataExpr} \\
\quad\mid & \mathit{Draw} & [\mathit{Picture}] \\
\quad\mid & \mathit{Anim} & [\mathit{Picture}]
\end{array}
$$

There are two kinds of compound picture; *Draw* combines a list of subpictures within one display frame, while *Anim* creates an animation, rendering pictures into successive frames. Novel combinations of time and space are possible, e.g. by composing slices from multiple timesteps into one frame, or animating a plane moving through a single timestep. *Picture* uses a small number of supporting definitions. For example, the *Range* type provides a vocabulary for sampled intervals:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Eq}\ a \Rightarrow \mathit{Range}\ a = \mathit{Single}\ a \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{Range}\ a\ a \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{Sampled}\ a\ a\ a
\end{array}
$$

It is used to specify the thresholds at which a scalar field is contoured or surfaced, and is also used to describe the spatial sampling of grids. The *Colour* data type specifies a number of schemes for mapping sample values onto colours, while *DataExpr* is used to select the field to be visualized, and includes support for derived fields. The real gain comes from the *embedding* of the DSL within Haskell, allowing us to generate animations using specifications such as:

```
overDensity =
  let slice t s = Use (From (Range 0 599) (Range 0 247) (Single 124) t s)
  in Anim  [ Draw [Slice mblues  (slice t D)
                  , Contour mgreens (Sampled 200 400 1000) (slice t Mv)
                  , Contour reds (Sampled 0 0.02 0.4) (slice t H2xD)
                  ]
           | t ← [5, 10 . . 195]]
```

This example creates an animation showing correlation between the shockwave (as captured by overall gas density D), turbulence (Mv), and the absolute density of H_2, captured by the derived field $H2xD$. Figure 4 shows a snapshot from the animation, revealing that H_2 formation (white) is concentrated in regions bracketed by the shockwave (blue) and higher-turbulence regions (green).

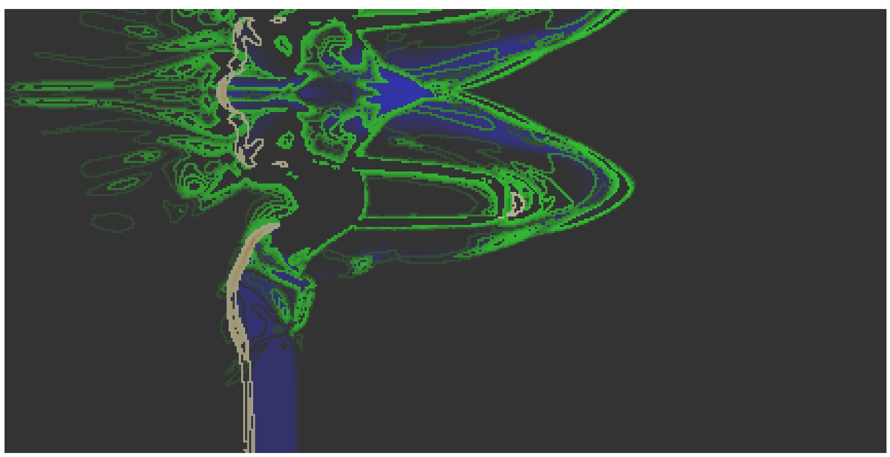

Fig. 4. Combination of gas density (slice), turbulence (green contours), and absolute H_2 concentration (white contours).

Evaluation of a DSL expression is carried out in the context of an *environment* that carries the various data grids referenced from within the expression. A *Picture* expression is interpreted by a function *eval_picture* that pattern matches each of the *Picture* constructors, extracts appropriate grids from the environment, and constructs a scene graph node to carry the visualised geometry. Here, for example, are the cases for *Contour* and the two compound picture types:

$$
\begin{aligned}
&\mathit{eval\_picture} :: \mathit{Environment} \rightarrow \mathit{Picture} \rightarrow \mathit{HsScene}\\
&\mathit{eval\_picture}\ \mathit{env}\ (\mathit{Contour}\ \mathit{pal}\ \mathit{thresholds}\ \mathit{dexpr})\\
&\quad = \mathit{Group}\ \mathit{static}\ \mathit{geomlist}\\
&\qquad \textbf{where}\\
&\qquad\quad \mathit{levels} \quad = \mathit{range\_to\_list}\ \mathit{thresholds}\\
&\qquad\quad \mathit{nr\_levels} = \mathit{float} \circ \mathit{length}\ \$\ \mathit{levels}\\
&\qquad\quad \mathit{field} \quad = \mathit{eval\_data}\ \mathit{env}\ \mathit{dexpr}\\
&\qquad\quad \mathit{plane} \quad = \mathit{slice\_plane}\ \mathit{dexpr}\\
&\qquad\quad \mathit{mkgrid} \quad = \mathit{squareGrid}\ (\mathit{cell\_size\_2D}\ \mathit{field}\ \mathit{plane})\\
&\qquad\quad \mathit{points} \quad = \mathit{mkgrid}\ \$\ \mathit{plane\_points}\ \mathit{dexpr}\ \mathit{field}\\
&\qquad\quad \mathit{values} \quad = \mathit{mkgrid}\ \$\ \mathit{samples}\ \mathit{field}\\
&\qquad\quad \mathit{colour} \quad = \mathit{transfer}\ \mathit{pal}\ 1.0\ 1.0\ \mathit{nr\_levels}\\
&\qquad\quad \mathit{contours} = \mathit{map}\ (\lambda t \rightarrow \mathit{concat}\ \$\ \mathit{isosurface}\ t\ \mathit{values}\ \mathit{points})\ \$\ \mathit{levels}\\
&\qquad\quad \mathit{colours} \quad = \mathit{map}\ \mathit{colour}\ [1.0\,..\,\mathit{nr\_levels}]\\
&\qquad\quad \mathit{geomlist} = \mathit{zipWith}\ \mathit{contour\_geom}\ \mathit{contours}\ \mathit{colours}\\
&\mathit{eval\_picture}\ \mathit{env}\ (\mathit{Draw}\ \mathit{ps})\\
&\quad = \mathit{Group}\ \mathit{static}\ \$\ \mathit{map}\ (\mathit{eval\_picture}\ \mathit{env})\ \mathit{ps}\\
&\mathit{eval\_picture}\ \mathit{env}\ (\mathit{Anim}\ \mathit{ps})\\
&\quad = \mathit{Animate}\ \mathit{anim\_control}\ \$\ (\mathit{True}, \mathit{map}\ (\mathit{eval\_picture}\ \mathit{env})\ \mathit{ps}, [\,])
\end{aligned}
$$

The brevity of the compound cases, *Draw* and *Anim*, is particularly pleasing. Constructors for compound *pictures* are interpreted directly in terms of an analogous *rendering* constructor acting on the interpretation of the sub-pictures. Composition of pictures is essentially an application of *map*. The only differences between the interpretations of Contour (2D) and Surface (3D) are (i) the *mkgrid* function for Surfaces builds a cubic grid, and (ii) the geometry is constructed by $surface\_geom$ rather than *contour_geom*.

6 Comparisons with other approaches

Previous entries to the visualization contest have used large-scale visualization tools such as VTK and Amira, and/or specialised graphics hardware. Our submission represents a radical departure. In place of a large scale visualization toolkit, we utilised a small, lightweight Haskell library running on a modest desktop PC. A direct comparison is difficult. Our solution consists of less than 4000 lines of Haskell and 630 lines of C, whilst for example VTK [3], a powerful toolkit for visualization developed over more than a decade, consists of nearly 1000 C++ classes, and 600K lines of code. Even comparing specific features, e.g. our isosurfacing implementation with VTK's is non-trivial; the VTK module has to deal with a more complex data and execution models, but excludes the machinery for building and executing pipelines, which arguably should be counted. Despite these caveats, this overall comparison, along with the figures presented in [10] do again highlight the brevity and expressive power that come with functional abstractions.

Brevity is particularly interesting in the context of *exploratory* visualization. Although we started with a number of algorithms already implemented, the contest tasks required new infrastructure and techniques. These were developed on the fly within the four weeks in which the authors were working towards an entry. Isosurfacing and volume-rendering code was reused, but interfaces for slicing and contouring were built, and animation, 3D scatterplot, and of course the fixed precision library and down-sampling infrastructure were all new. We would estimate that less than 1000 lines of Haskell were written or modified specifically for the contest. *The practical implication is that, when faced with a novel visualization problem, it may well be easier to write a new bespoke technique in 20-30 lines of Haskell than to assemble a collection of coarse-grained modules within a large toolkit, let alone create a set of new modules.*

There are two areas where our solution raises larger questions. First, we found it necessary to use C to implement data conversion and selection. A Haskell utility for converting the input data files into our binary fixed-precision format required ≈45 minutes per file. The C utility runs in less than 2 minutes per file. When processing 200 files, this is a significant difference. Haskell's support for generating tight, fast loops is not yet ideal. Second, we are uncertain as to the value of the fixed precision library. Our use of two 16-bit integers for mantissa and exponent involves a trade-off between precision and range relative to the 21-bit mantissa and 9-bit exponent of the IEEE Float type. Ultimately, all values are converted to float for rendering. Any advantages accrue from computation of derived fields, for example the vector-field curl, or the absolute H_2 density computed as the product of gas density D and relative H_2 proportion. However, beyond this one case study, scientific visualization involves extensive numerical computation, and it was notable how readily Haskell facilitated both the definition and testing of a new numerical type.

Our major success was the DSL for pictures, which gave us considerable freedom to explore the data. We are far from the first to realise the benefits of this approach in the context of graphics. 'Picture combinators' go back at least as far as Henderson's 1982 paper on functional geometry, recently revisited [15], and Arya's work on functional animation [16] provides a rich set of operators for constructing movies. More recently, Elliott has produced a series of papers showing the value of DSLs for image manipulation (Pan, [17]) and graphical synthesis (Vertigo, [18]). Our DSL was implemented only in the final week of the contest. Initially, we had concentrated on data management and visualization techniques. The 'forcing function' for change was the need to include animation. At this point we finally appreciated how much of a hindrance our ad-hoc construction of pictures was causing. With the DSL defined we were able to make rapid progress. Significant insights emerged literally within the final hour before submission. Even then, we were unable to fully exercise our system. We had for example implemented a 3D scatterplot, exploring in particular correlations between ion concentrations. Given our animation facilities, it would be interesting to create a time-varying scatterplot, showing how the relative concentrations evolve over time as the shock-front passes through space.

7 Conclusions and Prospects

This paper is not *just* about the use of Haskell for one specific problem, however challenging. The rationale for the IEEE visualization contest is to explore new approaches to challenging visualization problems, and the scenario explored here, in particular large volumes of multifield data, is one that is found widely in practice. Our contribution is to show how functional languages enable rapid exploration of new visualization techniques, and a particularly elegant way of describing novel *combinations* of technique.

The primitives of our picture DSL can be seen as analogs to the modules of a pipelined architecture [3]. However, we are working towards a different strategy. The contour code in Section 5.3 uses stream-based operations that generalise our initial work [9]. We would like to exploit these, and possibly a similar library on array-like structures, to provide an *intermediate* language for visualization algorithms. We see a visualization system as a hierarchy of languages. At the top, a declarative result specification (the picture DSL) is interpreted within a language of stream/array operations, which are then mapped onto a language for dataset management (cf our 'Stage I' as described in Section 4), generating datasets on demand, before finally a rendering language constructs scenes for display and interaction. Stages I and II would then be coupled directly, with the downsampler invoked directly from the visualization engine to provide datasets on demand.

The work presented here addresses *scientific* visualization. There is another challenge where functional programming may provide profoundly new insights, namely providing new levels of abstraction for managing *information* visualization (aka *infovis*). A key challenge here is the diversity of both data organization and visual metaphor. As a result, tools tend to be specialised to limited types of data and/or applications, and it is difficult to identify generic, reusable abstractions. The first task in any infovis application is to impose some structure on the data, one that enables translation into a suitable visual representation, for example a tree or graph. Could the strategy of creating layers of DSLs help also to structure infovis applications? An equally interesting question is whether the richer type system of functional languages, possibly including ideas like polytypism, can be used to find unexplored regularities within both data and display techniques, and then to exploit the connection between data and external representation via higher-level abstractions. Recent work [19] on using Haskell for *visual analytics*, a new synthesis of information visualization and statistical analysis, suggests that the conversation between functional programming and visualization has only just begun.

Source code for our implementation is available from the project web site, www.comp.leeds.ac.uk/funvis/

Acknowledgements

The work reported in this paper was funded by the UK Engineering and Physical Sciences Research Council.

References

1. McCormick, B., DeFanti, T., Brown, M.: Visualization in scientific computing. Computer Graphics **21**(6) (1987)
2. Haber, R., Lucas, B., Collins, N.: A data model for scientific visualization with provision for regular and irregular grids. In: Proceedings of Visualization91, IEEE Computer Society Press (1991)
3. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics. Second edn. Prentice Hall (1998) See www.vtk.org.
4. Rhyne, T.M., Tory, M., Munzner, T., Ward, M., Johnson, C., Laidlaw, D.: Information and scientific visualization: Separate but equal or happy together at last. In: Proc. Visualization, IEEE Computer Society Press (2003)
5. Whalen, D., Norman, M.: Competition data set and description. 2008 IEEE Visualization Design Contest http://vis.computer.org/VisWeek2008/vis/contests.html.
6. Borgo, R., Duke, D., Runciman, C., Wallace, M.: The 2008 visualization design contest: A functional dsl for multifield data. Contest Entry: Under review (2008)
7. Borgo, R., Duke, D., Wallace, M., Runciman, C.: Multi-cultural visualization : how functional programming can enrich visualization (and vice versa). In: Proc. Vision, Modeling, and Visualization, AKA Verlag - IOS Press (2006)
8. Borgo, R., Wallace, M., Duke, D., Runciman, C.: Generic idioms: A functional approach to scientific visualization. Computer Graphics Forum (2008) Under review.
9. Duke, D., Wallace, M., Borgo, R., Runciman, C.: Fine-grained visualization pipelines and lazy functional languages. Transactions on Visualization and Computer Graphics **12**(5) (2006) 973–980
10. Duke, D., Borgo, R., Runciman, C., Wallace, M.: Experience report: Visualizing data through functional pipelines. In: Proc. Intl.Conf. on Functional Programming, ACM Press (2008)
11. Spence, R.: Information Visualization. Addison Wesley (2000)
12. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: exhaustive testing for small values. In: Proc. of the ACM SIGPLAN Symposium on Haskell, ACM Press (2008) to appear
13. Wernecke, J.: The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor. Pearson (1994)
14. Courtney, A., Nilsson, H., Peterson, J.: The yampa arcade. In: Proc. Haskell Workshop, ACM Press (2003) 7–18
15. Henderson, P.: Functional geometry. Higher-order and Symbolic Computation **15** (2002) 349–365
16. Arya, K.: A functional approach to animation. Computer Graphics Forum **5**(4) (1986) 297–311
17. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming **13**(2) (2003)
18. Elliott, C.: Programming graphics processors functionally. In: Proc. of the Haskell Workshop, ACM Press (2004)
19. Heard, J.: A gentle introduction to functional information visualization. ACM SIGPLAN Developers' Track on Functional Programming (DEFUN) (2008) http://bluheron.europa.renci.org/docs/BeautifulCode.pdf.