A Language for Mathematical Visualization

John Peterson[1]

Yale University, Department of Computer Science
New Haven, CT 06520-8285
peterson-john@cs.yale.edu

Abstract. A domain-specific language (DSL) is a programming language adapted to the needs of a particular problem domain. A well designed DSL gives unsophisticated users the computational capabilities of an advanced programming language without the complexity of a general purpose one. We have applied DSL technology to an important new field: secondary education. In this paper we present a language that allows students to visualize mathematical concepts in subjects ranging from basic algebra to calculus.

We have adapted an existing DSL called Pan for use in the classroom. Pan is a language of functional images: pictures are represented by a mapping from points in the coordinate plane onto colors. Using the basic tools of functional programming, users can describe complex images clearly and succinctly. Pan allows basic mathematical concepts to be visualized in new and creative ways. These visualizations are interactive: students can adjust the parameters which define the image with a GUI attached to it. Using this language, computers can be tightly integrated into the core high-school math curriculum and provide new learning opportunities that mix the rigor of mathematics with artistic creativity. The simplicity of this language and its direct relationship with the underlying mathematics makes it a tool that all students can use - not just advanced or computer-literate ones.

In this paper we demonstrate how functional images can be used for mathematical visualization in the classroom and discuss implementation issues encountered when porting software to a secondary school setting. Some preliminary experiences in the classroom are also included.

1 Introduction

The use of computers in high school mathematics courses is often limited to primitive devices such as a graphing calculator. Beyond this rather minimal level of functionality there has been no consensus as to appropriate tools to support math education. Inspection of any math textbook, from algebra to calculus, reveals that visualization is an essential part of the learning process. These books are filled with graphs and pictures used to represent the abstraction ideas of mathematics in a visual form. The graphing calculator is a simple extension of these textbook visualizations: the primary purpose of these machines is to create pictures, not to calculate values. Our goal is to empower educators and students by using declarative programming language technologies to create a language-based visualization tool.

Why are graphing calculators so popular with educators? First, they are *general*: they can be used in many different contexts. The time spent teaching a student to operate such a device is rewarded by its frequent use throughout many different subjects. Second, calculators are *simple*. Unlike a general purpose programming language, the language used by calculators can easily be understood by students and instructors without extensive training in the process of computation. Our goal is to take the next step: making visualization software *expressive*. It is this expressiveness that expands the role of computers in the classroom and allows students to explore the full range of mathematical subjects in a new and creative manner.

A formal notion of language is an essential aspect of our approach. Language provides the expressiveness and, more importantly, basis for understanding that a student needs to master mathematical concepts. We augment the standard mathematical language with just enough computational devices to support interactive visualization. This in fact sums up the basic purpose of declarative programming: allowing the same language to be used at both the computational and conceptual level in an application. A good language is the software equivalent of a laboratory. A student of chemistry can, for example, learn much from freely experimenting with a well-stocked chemical lab. A language of mathematical visualization similarly supplies the basic elements of mathematics and allows a student to experiment freely with their combinations.

We believe that better visualization tools will have a significant impact on education. Reasons for this include:

- Computer-based visualizations can be *interactive*. That is, a GUI allows the student to control parameters in the visualization. This allows exploration a high dimensional space in a simple, intuitive manner.
- We wish to *customize* visualization to meet the needs of particular students or lesson objectives. There are many different ways to convey mathematical concepts visually; different students respond to different visualization styles.
- An expressive visualization language adds an element of *creativity* to the lesson material. Students that experiment with new styles of visualization are rewarded by the artistic aspect of the images they generate. This style of learning complements the traditional problem solving approach that predominates in mathematics.
- Using a formalized notion of *abstraction* we give users a way to expand their *mathematical vocabulary*.

To achieve the full benefits of a new educational tool, we must not only design the tool itself but also integrate it into the learning process. Thus our research must address curricular integration and assessment issues as will as language design and implementation.

Our project is still in a preliminary state. The software is not yet ready to deploy in the classroom but prototypes are available for preliminary assessment. We have performed limited classroom experiments but have not yet obtained a significant amount of experience working with students and educators. We believe, however, enough has been accomplished that sharing our research with the community will be of value.

2 A Language of Functional Images

This work is based on Pan[3], a language of functional images[1] developed by Conal Elliott. Pan is embedded in Haskell[7] and uses the Haskell type system and syntax. We have re-implemented Pan as a stand-alone system, independent of Haskell. We have altered the original Haskell-based Pan system in a number of cosmetic ways: a simplified syntax, a less complex type system, and a new backend, but the semantic core of Pan remains unchanged. Although the following examples use this new syntax and type system they can be translated into the original Haskell version of Pan with little effort. Since the essence of the Pan language is unchanged, we have not given our language a new name and we use the name "Pan" to refer to our new system while "Haskell Pan" is the original system.

2.1 Pan Basics

Programs in Pan are constructed using *definitions*. These place students on familiar mathematical ground: the idea of definition is fundamental to basic mathematics. Definitions, in turn, utilize *expressions*. These use standard mathematical notations (as much as possible) and are also familiar to students.

What is somewhat more foreign to students is the notion of types. Although types such as integers, reals, and points are quite familiar, formal type systems are not part of standard mathematics education. But since types are essential to understanding and correctness of definitions, we include these in Pan and expect that students can grasp simple type definitions. This aspect of the system is certainly subject to further investigation. Unlike the Haskell-based Pan system, we do not support type abstraction: the definition of new data types by students. At present we also do not have type declarations in the language; types are only seen in error messages and documentation.

An image is represented as a function from points to colors. In Haskell, this type would be represented as follows:

```
type Image = Point -> Color
```

This definition produces black and white checkerboard on a Cartesian plane with squares of 10 x 10:

```
checker (x,y) =
   if even(floor(x/10) + floor(y/10))
     then white else black
```

At a point (x,y), the image is white if the sum of the scaled integer parts of the coordinates is even, otherwise black. The names white and black are part of a sub-domain of colors.

A *lens* is a function from points to points, as captured by this Haskell type:

```
type Lens = Point -> Point
```

A lens can be used to distort an image such as the checkerboard. The following lens function, invertP, inverts the magnitude of a polar coordinate:

```
invertP :: Lens
invertP (r @ theta) = (1/r @ theta)
```

The (r @ theta) notation is used to represent points in polar coordinates. This is an example of *views*, a generalization of pattern matching. Although views are not part of Haskell we use them in Pan.

We can use this transformation to "warp" the checkerboard as follows:

```
  flower = checker . trans
```

Since images are represented by functions, function composition (the ".") operator) can be used to couple the transformation with the original image. This produces the picture shown in figure 1. Pan allows images to be constructed with

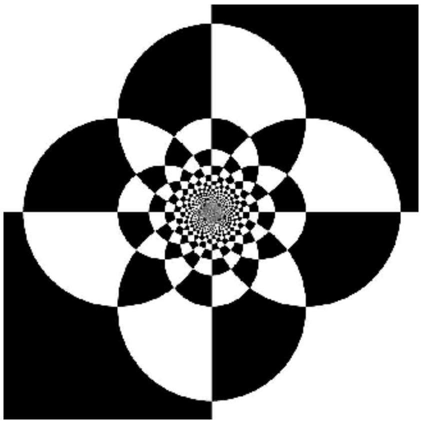

Fig. 1. A Warped Checkerboard

almost any degree of mathematical sophistication. This example demonstrates the conciseness of Pan programs: only a few lines of code generate a relatively complex and aesthetically pleasing image.

Controls can be added that allow interactive adjustment of image parameters. The following lens has two controls: an integer that determines the number of ripples and a ripple amplitude:

```
n              <- islider("n", 1, 20, 1)
amp            <- slider("amplitude", 0, 1, 0)
lens (r @ a) =  (r*(1+amp*sin(n*a)) @ a)
picture      =  checker . lens
```

Figure 2 shows the entire view window with the attached GUI. The menus along the top are standard for all images while the two sliders at the bottom are specific to this image. The parameters to the slider functions are the label, minimum, maximum, and initial values. Also note that the sliders allow students to view or directly enter the exact numeric values, enabling manual checking of the calculations that generate the picture. We use `<-` instead of `=` in the definition of the controls to indicate the creation of a user interface object instead of an ordinary definition. There should be precisely one copy of the controller even if the value generated (`n` or `a` is used more than once. That is, equational reasoning valid for `=` but not `<-`. This distinction is related to the monad used in the Haskell version of the system. This monad is explicated in our translation and remains hidden to student users.

2.2 Visualization Functions

A visualization function converts a value of some specific type into an image. There are usually many ways to view objects of a particular type. For example, a function from real to real may be visualized as a line on a plane, a region bounded by this line, an animated object moving in trajectory, a radial distortion of a picture, or even as a sequence of colors defined via a color mapping. Students and instructors are free to invent new visualization functions for any type of object. We anticipate building a large library of visualization functions into Pan when it is released. Standard visualization interfaces can include standard graphics visualization interfaces such as pan and zoom capabilities.

The following visualization function is used in the next section to view lens functions. The user selects a picture using the `selectImage` GUI and the lens is used to distort the picture. Imported pictures are represented in the same manner as the generated picture using an image function. A set of controls allows the user to move the picture around under the lens, allowing the student to adjust the result in a number of ways. The function `f` may contain additional controls; the final GUI combines controls from both the the visualization function and its target.

```
im <- selectImage("Image to warp")
s  <- slider("Scale", 0.1, 50, 1)
x  <- slider("X motion", -200, 200, 0)
y  <- slider("Y motion", -200, 200, 0)
r  <- slider("rotation", 0, 360, 0)

showLens(f) = im' . f
  where
```

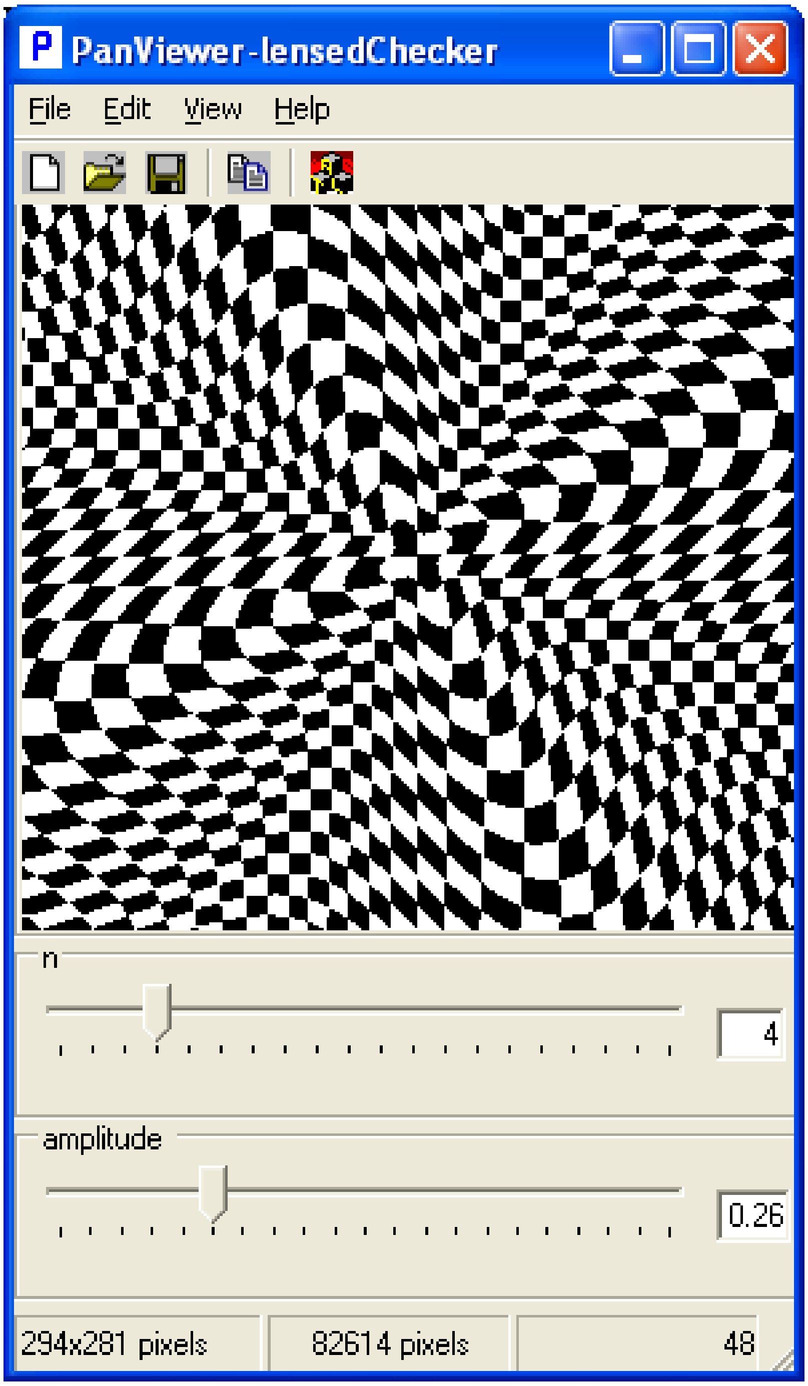

Fig. 2. An Image with Controls

```
im' = rotate(r,
        translate((x, y),
          scale(s, im)))
```

This example demonstrates the simplicity of constructing new viewers. An instructor can build specialized viewers for a lesson, allowing complete control over the visualization environment.

Pan programs are written using a small development environment. This has two editable windows: one containing an unordered set of definitions and the other containing an expression to be visualized. The user interface provides a simple "viewer manager" in which object types are associated with default viewing functions, giving students an implicit way to visualize objects of a given type. Figure 3 shows the development environment. At present we are working on a

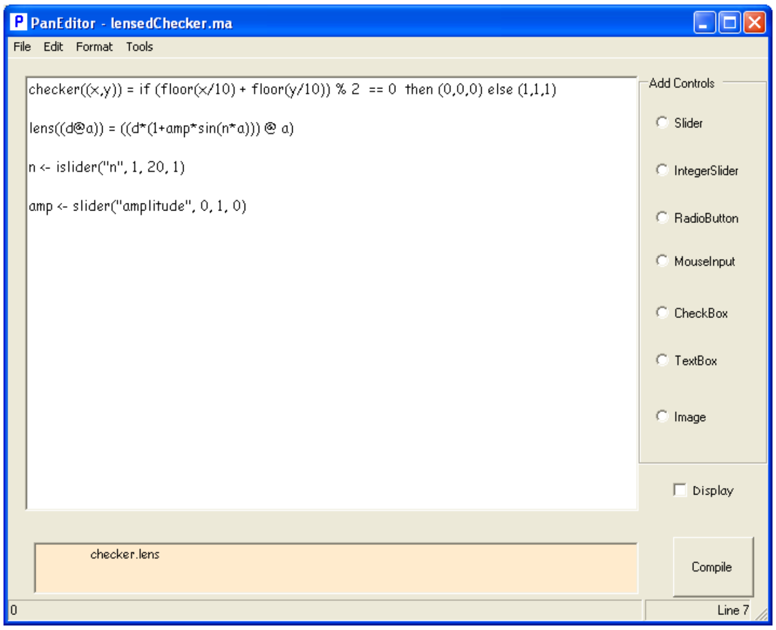

Fig. 3. Writing a Pan Program

web-based interface to Pan that will allow students to run the system remotely without installing any software.

3 Experiences

We have performed two experiments at using the Haskell Pan in the classroom. The first was in an eighth grade algebra class and the second was for high school junior and seniors. The slides and galleries from these presentations is available at `http://haskell.org/edsl`, our website of educational DSLs. Each of these presentations used only one viewer: the lensing function defined in the previous section. Each commenced with a group lecture of about 45 minutes that covered the following topics:

- The definition of functional images.
- Lensing functions: the effect of slope (this determines locally whether the image is expanded or reduced, mirrored or non-mirrored), function ranges and domains, and the effect of local minima or maxima.
- Polar coordinates and simple polar transformations.
- Simple periodic functions such as the sawtooth.

After the presentation, students were asked to invent a new lensing transformation and sketch its anticipated effect. The next day these transformations were entered into Pan under the supervision of the presenter and each student created a picture for a picture gallery. Figure 4 was generated by a student in an eighth grade algebra class; it uses a polar transformation that swirls an image of Thomas Edison. The code generating this image is as follow:

```
k1 <- slider("k1", 0, 100, 0)
k2 <- slider("k2", 0, 10, 0)
f (d @ a) = (d+k1 @ a+k2*(d+k1))
```

Figure 5 was created by a high school senior using radial distortion; the code for this is:

```
k <- slider("k", 0, 50, 0)
p <- islider("lobes", 0, 20, 1)
f (d@a) = (d/(1+k*(abs (sin (p*a))))@a)
```

After the presentations, we obtained feedback from the math instructors. Their observations include:

- Functional images were an excellent way to illustrate the basic idea of a function. Since the domain and range are captured visually it made these concepts particularly clear.
- Functions from 2-D point to 2-D point were somewhat new to the eighth grade (14 year old) students and served to re-enforce their basic understanding of functions. The textbook used by the students used mostly lines on a coordinate plane to visualize functions so warping functions were new to many students.
- The mathematics was very "close to the surface". That is, it was possible to precisely explain why the results appeared as they did using only the mathematics in the lesson. This is in contrast to students who were using Photoshop to alter photos without understanding the mathematical basis for the warping effects.

Fig. 4. Warped Edison

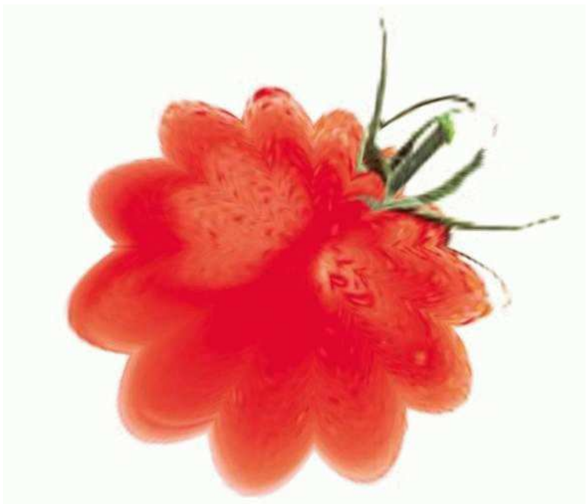

Fig. 5. Splatted Tomato

- More than the expected number of students volunteered to do the follow-on assignment (the "define your own lens" part), indicating a unusual level of interest.
- The better students (calculus level pupils) were adept at inventing equations that would have a desired effect. This activity turned into a challenge among the students in which one would describe an effect and another would write down some equations.
- The use of a GUI to adjust parameters gave the students a very good intuitive feel for the components of the equations used. This was especially true for periodic functions where it was easy to demonstrate the ideas of period and phase shift through interactive controls.

These initial demonstrations of Pan were very encouraging; students and instructors were enthusiastic about the potential of this software.

4 Implementation Issues

The Haskell version of Pan is implemented in a unique way: Pan programs are Haskell programs that generate what is essentially an abstract syntax tree for a simple functional language; this is then compiled with much optimization into C++ code. This C++ code is then dynamically loaded into a viewer that creates the GUI associated with the image and renders the image to the screen. The speed of the C++ code is quite important: as students change image parameters using the GUI the system should be able to update the image without delays.

The Haskell version of Pan is unsuitable for direct student use: it requires an intimate knowledge of Haskell and its implementations. Type errors are particularly difficult to deal with and in our experiences with the original Pan in the classroom showed that a Haskell-knowledgeable supervisor was essential. To create a stand-alone system, we have made a number of significant changes and additions to Pan. These include the following:

- We have written a custom parser that has better error handling than current Haskell compilers. Since many advanced language features have been removed it is much easier to identify parsing errors.
- A feature Haskell lacks turns out to be quite useful in this domain: views. We use views to handle pattern matching in either polar or rectangular coordinate systems. The set of views is pre-defined at present - students cannot yet define new views.
- The type system is mostly hidden from the user. We compute standard Haskell types during type checking but hide parts of the signatures from the user. This avoids revealing the difference between static and dynamic values and the use of the `UI` monad.
- As has been observed in Dr. Scheme, error handling is perhaps the most important aspect of the compiler. We have designed our software with this in mind and expect to continue to improve this aspect of the system over time. Many of the problematic errors that plague Haskell (kind errors, the

monomorphism restriction, overloading problems) have been designed away in our language.
- In Pan, the user program is de-functionalized "for free": all functions are expanded away by the Haskell system before they reach the Pan compiler. Here we must perform this expansion using a small lambda calculus interpreter. One advantage of our approach is that we can catch compile-time loops using iteration limits in cases where the original Pan system would generate stack overflows.
- The run-time language has been changed to C♯ to make the system more portable. The image viewer has also been ported to C♯.
- The display window can contains both an image and program generated text. This allows students to easily incorporate extra information in the output of the system. This can be used to provide feedback related to the current mouse position, allowing the mouse to interactively sample image parameters or other values that would aid the student in understanding the generated image.
- We have defined a simple module system that allows students to access code libraries. An instructor may to provide visualization or other functions tailored to a particular lesson as a module for student use. This is much less complex than the Haskell or ML module systems.
- We are targeting a number of new execution platforms to make it easier to deploy this language. These include web servers and tablet computers.

These changes have been a significant undertaking. We conjecture that other educational languages derived from existing embedded DSLs would require similar efforts.

5 Related Work

Programming languages have been used with varying degrees of success in secondary education. Although languages such as Scheme, Java, and Basic can be used in a manner similar to Pan, a general purpose language can not approach the performance and simplicity of Pan in this application. We doubt any general purpose programming language could or should be used directly in a class that is not intended to teach programming itself. Declarative, domain-specific languages are the only possible way to incorporate the necessary functionality into an application such as this. Although we do not use Scheme itself, the Dr. Scheme[2] environment is a major source of inspiration regarding implementation and teaching methodology.

In spirit, our system is related to Mathematica in it's attempt to directly represent mathematical objects. Mathematica is an excellent vehicle for visualization and has many features far beyond what we have implemented in Pan. However, besides the size and price of this application, Mathematica does not have a sufficiently declarative language base to support the type of abstraction that is natural in Pan. While Mathematica is good at representing the mathematical objects for visualization, it is more difficult to create new visualization styles. It also lacks a sophisticated type system.

This work does not address the sort of visualization needed in most geometry lessons. In geometry, the emphasis is on proofs and constructions rather than functions; geometry domains tend contain discrete objects rather than continuous functions. There are a number of good software packages that visualize geometric information such as the Geometer's Sketchpad[4] or GeomView[5]. These systems use a a different set of tools: constraints, 3D viewers, and symbolic reasoning. The scripting language used by the Geometer's Sketchpad is quite declarative and expressive; it is perhaps closest in spirit to Pan. There is a wealth of resources available for it, including examples, lesson plans, and activities. We conjecture that functional programming can be integrated into this setting but have not pursued this yet.

Another related area is modelling and simulation. Here the primary declarative language is Modelica[6]. Although this language does not have the abstraction capabilities of a functional language such as Pan it could be used to handle physical simulation in science education. The computational model underneath a simulation language is considerably more complex than that which Pan is based on. Simulation is inherently stateful: as time moves forward the state of the system changes. This requires a significantly different runtime engine.

The relationship between this system and systems based on symbolic reasoning is illuminating. We do not manipulate the formulas entered by our users in any user-visible way - we simply compile them into visualizations. Tasks such as symbolic differentiation or equation solving are not part of Pan's repertoire. This does not preclude development along these lines: the language we use to represent mathematical objects is quite suitable for this purpose. Equational reasoning would provide the formal semantic framework for such symbolic manipulation.

6 Conclusions and Future Work

We have demonstrated a language-based visualization system that enables students to explore general mathematical concepts in new and interesting ways. This research has the potential to bring powerful computer-based tools to the traditional high school mathematics curriculum. Using a language that fits seamlessly into the existing educational context, students can put mathematical knowledge to work constructively, building new and unique images based on a wide range of mathematical concepts.

The next challenge is to integrate this language more directly into the curriculum. We plan to develop a set of lessons in subjects from basic algebra to calculus that incorporate student directed visualization.

Serious research into educational methodology must include detailed assessments of new educational techniques. We are working with the Pace group at Yale to design rigorous psychological tests and use them to refine our system. We expect these tests to provide a wealth of feedback on our language and software. Using a properly designed test we can measure the impact of our approach on basic mathematics knowledge as well as less tangible capabilities such as creativity.

Ours approach is not limited to mathematics education. Science instruction can also benefit from declarative language methodology. We are investigating languages for modelling and simulation that would aid students understanding physical systems.

7 Acknowledgements

We would like to thank Conal Elliott for his assistance and Microsoft Research for supporting this project. Thanks also to the Pan team: Jian Yuan, Emmanuel Imbeah, and David Eisenstat

References

1. K. Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, 1994.
2. The dr. sceme project. `http://www.drscheme.com`, June 2002.
3. Conal Elliott. Functional images. Technical report, Microsoft Research, 2001.
4. The geometer's sketchpad. Key Curriculum Press, `http://www.keypress.com`, June 2002.
5. Geomview – an interactive 3d geometry viewing program. The GeomView Team, `http://www.geomview.org`, January 2002.
6. Modelica – a unified object-oriented language for physical systems modeling: Language specification version 1.4. The Modelica Association, `http://www.modelica.org`, December 2000.
7. Simon Peyton-Jones. Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 98). *www.haskell.org*, 2001.