Creating Beautiful Diagrams Using Language

Katherine Ye and the PENROSE team, *Carnegie Mellon University*

Typesetting software like LaTeX has accelerated scientific communication by beautifully *typesetting* plain-text notation, but no equivalent exists for making *diagrams*. We are therefore building a system called PENROSE, which allows people to **create beautiful diagrams by just typing mathematical notation in plain text**. Penrose aims to enable non-experts to create and explore high-quality diagrams, providing deeper insight into challenging technical concepts. Figure 1 illustrates how abstract statements about concepts in linear algebra may be automatically converted into a diagram. (I will use linear algebra as a running example, but PENROSE is designed to be extensible to any mathematical domain.)

PENROSE is a highly interdisciplinary project that **draws on insights from the areas of programming languages, computer graphics, and systems.** The core challenge in designing the PENROSE languages is to provide customizable visual representations for concept-level expressions in an extensible collection of domains. The core challenge in designing the the numerical solver is to quickly create diagrams with user-defined shapes, objectives, and constraints so these diagrams are both beautiful and meaningful. And the core challenge of building the whole system is to bridge the two: to automatically turn the conceptual relationships defined at the language level into a concrete visual representation. Rather than focusing on a fixed set of mathematical objects, we aim to make this framework user-extensible.

```
VectorSpace U, V, W, X
LinearMap f : U → V
LinearMap g : V → W
LinearMap h : W → X

Vector u1, u2, u3, u4, u5 ∈ U
Vector v1, v2, v3 ∈ V
u3 := u1 + u2
v1 := f(u1)
v2 := f(u2)
v3 := f(u3) = v1 + v2
Scalar a := det(u1, u3)
u4 := a * u2
u5 := -u4

Scalar m := |v2|
Scalar c := <v1, v3>
Vector w1 ∈ W := g(v1)
Vector x1 ∈ X := h(w1)
```

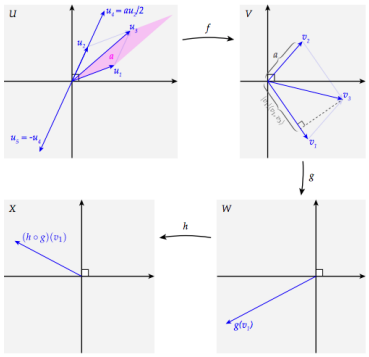

Fig. 1: Example notation and mockup diagram in linear algebra.

1 PENROSE makes it easy for anyone to make domain-specific diagrams

To address the key research challenges discussed in the introduction, we have built PENROSE to fit three design goals: to provide a *general-purpose*, *high-level*, and *extensible* platform for creating diagrams. What system design supports these goals? A compiler-like pipeline organization, shown in Fig. 2, is a natural choice. It fits the step-by-step construction of a diagram from textual user input, while allowing customization of each stage to support a highly expressive and usable system. We have designed PENROSE to comprise three languages for expressing diagrams, along with a compiler, optimization runtime, and user interface.

The first language, called the *domain description language* (DDL), provides a way for domain experts to concisely inform PENROSE about the types of objects in a domain and the relationships between them. Then, given a domain description, PENROSE creates two languages specifically for that domain. **The two languages, SUBSTANCE and STYLE, enforce a clean separation between content and form, akin to the separation between content and form provided by HTML and CSS.** SUBSTANCE is a programming language for specifying domain-level meaning, completely independent of visual representation; for instance, an expression in standard mathematical notation. STYLE is a language for defining how domain-level meaning is translated into a visual representation.

Given a domain description, a SUBSTANCE program, and a STYLE program, the PENROSE compiler synthesizes a constrained optimization problem corresponding to an objective function defining the "goodness" of the diagram. The optimization problem is solved by a numerical solver, yielding several beautiful and automatically-laid-out diagrams in a matter of seconds, which the user may then tweak by direct manipulation.

The generality of this approach offers many advantages. For example, all domains embedded in PENROSE have consistent syntax and semantics for their languages, all language implementers can tap into the general and powerful optimization runtime that the platform offers, and any improvement to PENROSE as a whole

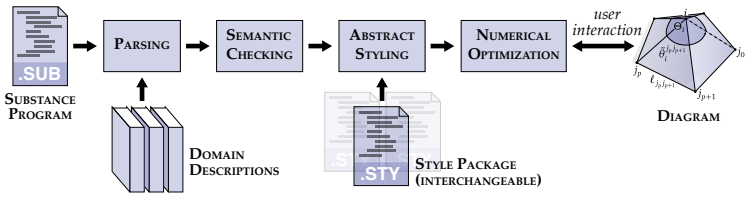

Fig. 2: The pipeline used to convert SUBSTANCE and STYLE programs to diagrams.

(*e.g.* a better layout algorithm or an expanded library of shapes and objectives) will automatically improve all domains embedded in the platform. PENROSE's language-based approach also offers great flexibility and generality. Most users of our system *will not require any graphic design skill to create beautiful diagrams*. The widespread success of LaTeX is a clear indicator that this kind of language-based paradigm can enjoy wide adoption. Just as LaTeX revolutionized written technical communication by codifying the best practices of professional typesetters (allowing authors to focus on content), PENROSE aims to revolutionize the way people visually communicate logical ideas.

In the following sections, I discuss **how the PENROSE pipeline works with a simple end-to-end example drawn from the domain of linear algebra, using real PENROSE code, highlighting the key research challenges at each stage**. Further information on PENROSE may be found in our OBT 2017 [1] and DSLDI 2017 [2] papers.

1.1 Domain implementers can use the domain description language to easily add domains

The domain description language (DDL) enables expert users to inform PENROSE about the meaning of a new domain. In a domain description, the user declares the abstract objects in a domain as types, declares functions that can be applied to them in terms of type signatures involving the objects, and declares logical relationships between objects as typed predicates. The DDL also enables users to declare custom syntax for the domain, which can automatically be used in SUBSTANCE and STYLE programs. The DDL's design builds on work in language workbenches like Spoofax [3] and programmable syntax [4], and we are extending it so users can define *composable* syntax extensions for mathematical notation in multiple domains.

```
tconstructor VectorSpace : type
tconstructor Vector : type
predicate In (v : Vector,
              V : VectorSpace) : Prop
operator AddV (v1 : Vector, v2 : Vector)
       : Vector
StmtNotation "Vector a ∈ U" →
             "Vector a; In(a,U)"
StmtNotation "v1 + v2" → "AddV(v1,v2)"
```

Fig. 3: A simple domain description for linear algebra.

Fig. 3 gives a simple example of how a language implementer might start to write a domain description for linear algebra in PENROSE. First, they define the *types* in the domain, here vector spaces and vectors. Next, they define a *predicate* that relates the objects in the domain at an abstract level; namely, that a vector may be an element of a vector space. Then, they define an operation that may be applied on objects in the domain; that is, two vectors may be added to yield another vector. Lastly, they define syntactic sugar for naturally stating concepts in the domain, such as adding vectors and stating that a vector is in a vector space. Using the domain description in Fig. 3, PENROSE automatically generates the SUBSTANCE and STYLE languages for making statements about linear algebra concepts and for defining visual representations of those concepts.

Types are central to PENROSE's language design. Building on the approach taken in [5], we model mathematical relationships in terms of types and predicates, which enables us to design clean mechanisms for extensible syntax, SUBSTANCE typechecking, and STYLE pattern-matching.

```
VectorSpace U
Vector u1, u2, u3, u4, u5 ∈ U
u3 := u1 + u2
u5 := u3 + u4
```

Fig. 4: A simple SUBSTANCE program for vector addition.

1.2 End-users can state high-level concepts in the SUBSTANCE language

SUBSTANCE is designed to model abstract, natural mathematical language in an extensible manner, inspired by the mathematical languages embedded in software like *Coq* and *Mathematica*. **Because SUBSTANCE has been formalized as a programming language, PENROSE can provide the end-user with powerful type inference and typechecking of mathematics.**

Consider two kinds of end-users: a student is learning about vector addition and wants to understand if this operation is commutative and associative, and a professor wants to illustrate the properties of vector addition in a textbook. These end-users will simply import the existing domain description and write natural mathematical statements just like what would appear in an introductory textbook, as in Fig. 4, which gives the addition of three vectors. Writing concepts in SUBSTANCE liberates the end-user from having to decide the visual appearance of objects in the domain or figure out low-level details like their positions and sizes.

1.3 Domain implementers concisely define visual representations in the STYLE language

The STYLE language, which builds on the design of stylesheet languages like *CSS* and *SASS*, enables a language implementer to define a *compositional visual semantics* for SUBSTANCE language in their domain. To do so, the user writes STYLE selectors and blocks that declaratively define a minimal set of shapes and spatial relationships that must hold to faithfully visualize the mathematics, *without having to* specify low-level details like position or size, and with the ability to tap into a powerful optimization runtime for automatic layout. **The semantics of STYLE itself includes a novel semantics for type-driven pattern-matching.** Most end-users will import a STYLE package that defines the visual representation of their domain, so I will describe how a STYLE package defines one common visual representation for linear algebra. (However, any STYLE package only defines *one* visual representation of a domain; one can easily import another style.)

```
Vector v
with VectorSpace U
where v ∈ U {
  v.shape = Arrow {
     start = U.shape.center
  }

  ensure contains(U.shape, v.shape)
  encourage nearHead(v.shape, v.text)
}

Vector u
with Vector v, w; VectorSpace U
where u := v + w; u, v, w ∈ U {
  u.shape.end = v.shape.end + w.shape.end

  u.slider_v = Arrow {
     start = w.shape.end
     end  = u.shape.end
     style = "dashed"
  }

  u.slider_w = Arrow { ... }
}
```

Fig. 5: Styling individual vectors and vector addition.

The first *selector* in Fig. 5 pattern-matches on all vectors in the SUBSTANCE program that are declared to lie in a vector space, and its *block* defines the visual representation of vectors as rooted at the origin of that vector space, which is drawn as a 2D Cartesian plane. (Styling for vector spaces is omitted.) The keywords **encourage** and **ensure** enable a STYLE writer to declare *objectives* and *constraints* that hold on the shapes, which are automatically solved by the runtime. Vector addition is then styled by refining the vector style. The second selector in Fig. 5 pattern-matches on all vectors declared to be the sum of two vectors, and its block styles them in the common "tip-to-tail" manner by performing the vector addition, also drawing "slider" vectors to show the "tip-to-tail" mnemonic.

1.4 Optimization solves the hard problem of automatic, general-purpose diagram creation

We have now seen a description for the linear algebra domain, written a SUBSTANCE program for adding three vectors, and found a STYLE package for the domain. How does all that text get turned into beautiful pictures? **When a user writes a STYLE program, they could be using any combination of custom shapes, together with any combination of custom objective functions and constraints defined on the shapes. Thus, the PENROSE system must provide an automatic, general, and extensible solver and layout engine.**

First, to perform optimization on arbitrary shapes, PENROSE must be able to quickly answer queries about shapes, *e.g.* distance, tangency, and intersection. Thus, we are designing fast algorithms for performing general shape-shape queries via polygonization. Next, PENROSE must be able to automatically optimize shapes' attributes with respect to these queries. Most optimization methods require taking the gradient with respect to the numerical parameters, so **the PENROSE pipeline is designed to be end-to-end differentiable**, even through arbitrary user-defined shapes and functions. Lastly, the optimization problem itself can be arbitrarily hard to solve: nonlinear, nonconvex, highly constrained, and composed of heterogeneous objectives (*i.e.* energy functions that behave very differently from each other). How can we ensure that the optimizer reliably solves these problems, and does so quickly, while yielding maximally beautiful and "useful" diagrams for the user? Currently, the PENROSE solver finds local minima of the energy function using a combination of gradient descent, line search, and the exterior point method. To improve the generality of the optimizer, we are designing methods for automatically improving the conditioning of the optimization problem, *e.g.* by normalizing the energy functions.

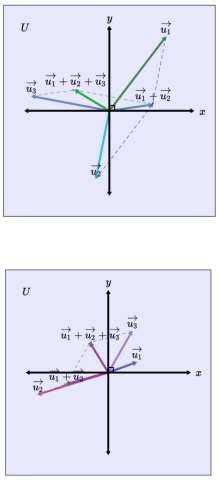

Fig. 6: Real PENROSE diagrams illustrating vector addition.

In our linear algebra example, the PENROSE compiler analyzes the combination of programs to create the initial state of this diagram, which consists of arrows, boxes, and labels. The compiler also builds the constrained optimization problem that is programmatically defined by the STYLE package, which consists of constraints requiring that every vector's shape be visually contained in the vector space's shape, as well as objectives encouraging every vector's label to be near the arrowhead.

Each solution to the optimization problem is a diagram. Two such solutions are displayed in Fig. 6. The solutions respect the meaning of the mathematics, illustrating that vector addition is indeed commutative and associative. The solutions also explore the "degrees of freedom" of parameters left unspecified in the mathematical notation. One interesting case is shown, where two parallel vectors are added.

The user may now directly manipulate the diagram (*e.g.* by dragging a label or arrowhead), in which case the optimizer can produce an adjusted diagram that still satisfies the constraints. The ability to specify graphical constraints, and the ability to manipulate the results while respecting these constraints (à la the software *Cinderella* [6]), frees PENROSE users from much of the tedious manual manipulation typical of existing illustration systems, while giving experts the detailed control needed to ensure quality output.

References

[1] K. Ye et al. Designing extensible, domain-specific languages for mathematical diagrams. *Off the Beaten Track*, 2017.

[2] K. Ye* and W. Ni* et al. Substance and style: domain-specific languages for mathematical diagrams. *Domain-Specific Language Design and Implementation*, 2017.

[3] L. Kats and E. Visser. The spoofax language workbench. In *ACM SIGPLAN Notices*. ACM, 2010.

[4] C. Omar. *Reasonably Programmable Syntax*. PhD thesis, Carnegie Mellon University, 2017.

[5] Mohan Ganesalingam. *The language of mathematics*. PhD thesis, Springer, 2010.

[6] Jürgen Richter-Gebert and Ulrich H Kortenkamp. *The interactive geometry software Cinderella*. 1999.