Construct:
Programming with Geometry

Sam Gruber

Advisors: Brad Myers, Stephanie Murray

Carnegie Mellon University

May 9, 2014

Chapter 1

Introduction

A programming language is a tool to help the programmer express a solution to a problem. This tool is most effective when it can clearly translate the intentions of the programmer into the machine's execution process. In most modern programming languages, a program is composed of linear sequences of text symbols. In domains such as text processing or algebraic computation, there is a clear correspondence between this representation of tasks and the way that the programmer imagines them.

However, there are many applications where such a format may not correspond well with the programmer's internal model. When the purpose of a program is to manipulate a geometric object, it is a peculiar burden to place upon the programmer to translate the intuitive spatial model of the program into a textual form in order to convey it to a computer. A similar disconnect manifests when parallel or concurrent programs are written in a text format that is still fundamentally singular and sequential.

Programmers should have the ability to create programs using tools that can represent the meaning of the program as clearly as possible, rather than having to jump through layers of translation. Enabling programming in these ways can open the field to the contributions of less-conventional participants, and could make possible new and different understandings of problems and solutions in computer science.

1.1 Goals

The target audience for Construct is visual professionals, such as architects and designers, who wish to apply programmatic tools to their existing process. This audience does not necessarily have experience with modern programming systems, but nonetheless is looking for a way to automate the generation of geometric forms from other geometric forms. This audience is comfortable thinking about objects in geometric terms, but may not be accustomed to working in symbolic terms.

Construct is designed to bring programming to this audience, where traditional programming often does not correspond to the programmer's mental model. We target four goals for the design of Construct to help programmers work with the language in a more direct and intuitive way.

- **Make clear the program state.** The programmer should be able to directly see the state of the program in the programming environment, rather than be required to mentally model the execution of a complex system.
- **Work in the native representation of geometric relationships.** Data and logic which is fundamentally geometric in nature should be expressed geometrically, rather than symbolically. The programmer should not be required to internally translate the meaning of the program to explain it to the computer.
- **Specify relationships without inference.** The programmer should be able to clearly and unambiguously present the program to the computer. Inferencing prevents the programmer from clearly understanding what will actually happen when the program runs.
- **Run a full (Turing-complete) range of programs.** The medium of programming should not prevent the programmer from expressing any possible program, though it may be better suited to producing certain types of programs.

1.2 Related Work

The use of computer systems to manipulate geometry has been an active area of research since Ivan Sutherland's work on the Sketchpad system [10]. Greg Nelson's later work on Juno [6] and commercial development of computer-aided drafting and design software has focused on the application of constraints to direct manipulation drawing interfaces. These systems have been valuable tools to aid manual acceleration of geometric work, but fundamentally are only an enhanced version of pen-and-paper geometric techniques.

Bret Victor has recently spoken [11] on the value of wide-ranging approaches to computer science and programming. Recent work by Victor [12, 13] has focused on the potential of visual tools which enable the development of systems with behavior. Victor's work engages with a data-centric view of programs, which is effective for representing compositions, but does not show the evaluation structure of the programs that are being composed.

At the same time, there have been explorations into computer architectures suitable for work in geometric terms. Tony DeRose's analysis of coordinate-free geometric calculations [2], though envisioned in the context of traditional text-based programming languages, provides a model of geometric computation that more closely corresponds to pen-and-paper geometry. Research into the Geometric Machine [8] developed a computer system in which the memory is envisioned as a geometric space rather than the familiar Turing-machine one-dimensional stream of symbols.

Construct aims to synthesize constraint-based geometry, visual tools for programming, and geometric bases for computation into a platform that engages visual and spatial thinkers and enables novel approaches to problems in computer science.

Chapter 2

Geometric Syntax

Construct proposes a new geometric method for creating programs. Most existing languages offer a textual syntax for composition, which offers an essentially linear approach to writing programs. A program in Construct is created by associating geometric objects and relationships in a 2d space. This allows the programmer to explore the design of a program in a more open manner, and facilitates the expression of programs which need not have a linear evaluation order.

Furthermore, a geometric syntax provides a format that can easily express problems which have a geometric or visual interpretation. Points, lines, and circles are all primitive objects in Construct, and are presented to the programmer visually, harnessing the brain's highly evolved visual reasoning systems.

While it is possible to perform symbolic computations in Construct, they are not the target problem set. Therefore, a programmer might find certain symbolic tasks, such as text manipulation, as roundabout in Construct as geometric tasks are in a textual syntax.

2.1 The Programming Environment

Because Construct does not use a textual syntax, programs cannot be composed in a text editor. Rather, they require an environment specifically suited to geometric programming. This section describes the fundamental characteristics of such an environment and presents a design for its interface.

Figure 2.1 shows a general design for the interface, highlighting the five important regions:

(A) Menu Bar
The Menu Bar provides access to functionality which has no representation in the language of the program itself, such as loading or saving programs or starting a live interpreter.

(B) Tool Palettes
Tool Palettes contain commands which create new elements that have meaning in the program. The three main palettes provide commands for Instantiation (see Section 2.2), Definition (see Section 2.3) and Modification (see Section 2.4).

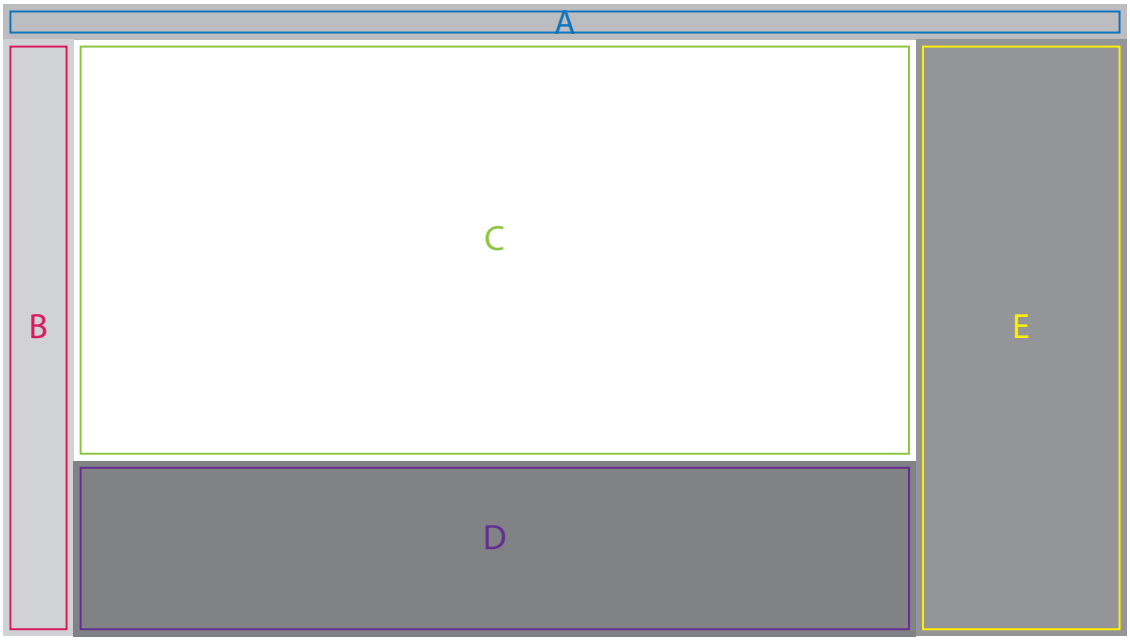

Figure 2.1: An interface diagram of the programming environment

(C) Program Space
The Program Space shows a data view of the Construct program. It allows the the programmer to view directly how the program will behave, using representative examples for objects which are not precisely defined. This programming-by-examples approach mimics the manner in which geometry is often taught and worked with, reducing the mental load on the programmer to imagine the meaning of the geometric relationships.

(D) Program Graph
The Program Graph shows the dependency relationships between the objects in Construct. Since all definitions flow from a group of predefined objects to an-as-yet-undefined object, the Program Graph produces a directed acyclic graph (DAG).

(E) User Content Drawers
The User Content Drawers contain definitions which have been created by the programmer as abstractions. The programmer may use these definitions in a similar manner to using the built-in definitions.

2.2 Object Instantiation

To begin working with any of the objects in Construct, the programmer must first *instantiate* them. One tool palette in the programming environment provides commands that instantiate new objects.

When first instantiated, a new Construct object is undefined. A point could have any position in the two-dimensional space of the program. A line object could have any position and any

orientation. A circle could have any center position and any radius. Distances and angles can have any size. A set could have any number of member objects.

Also notice that the instantiation operation does not require the programmer to name the object. In a textual syntax, names are necessary in order for both the programmer and the parser to understand what objects are being used at any stage of the program. In Construct, the programmer can simply *see* the difference between objects. The system, similarly is free to reference objects directly, since the programming environment can provide an unambiguous description of which objects are being used. Naming objects is supported in Construct, for the convenience of the programmer, but the names are simply discarded for the purposes of execution.

2.3 Applying Definitions

In order for objects to have any meaning, the programmer applies definitions to them. These definitions establish relationships between the objects in a Construct program. Since definitions are relationships between objects, almost all of them require multiple objects to be instantiated in the program space. Section 3.3 lists the definitions which are built into Construct.

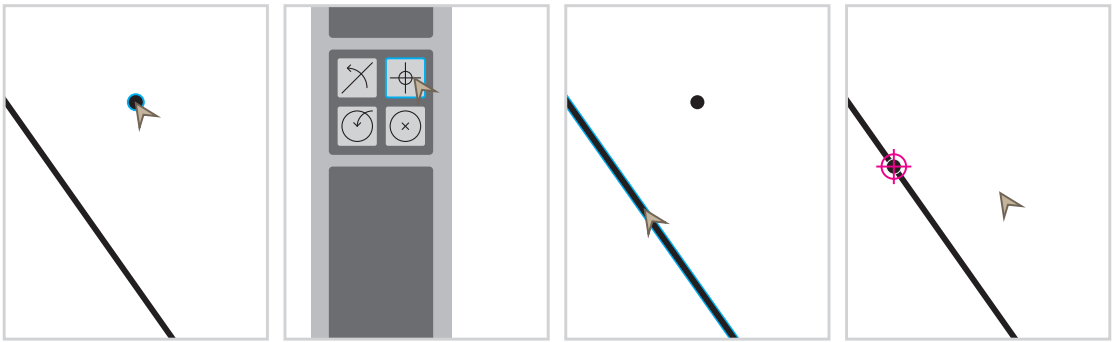

Figure 2.2: The interaction to apply a definition to an object

Applying a definition is an infix operation in the programming environment. First, the programmer selects the object which will be defined. Then, the programmer selects a definition from a tool palette in the programming environment. Finally, the programmer selects the objects which are referenced by the definition. Figure 2.2 shows how these interactions may appear in the programming environment, using the `on` definition as an example.

We can imagine a similar statement in a textual language:

```
Point x = arbitrary_on_line(l);
```

The effect at runtime of executing this part of the program may not be as linear as the textual equivalent suggests. In most cases, the object which is being defined will be updated to conform to the new definition's requirements, and other data (if not dependent on the object being defined) will not change.

However, in some cases, the new definition constrains the object in a way that is unsatisfiable. In these circumstances, the rest of the program will be recalculated in an attempt to satisfy the new definition. If this is successful, the programmer may see many objects in the program space rearrange. If it is unsuccessful, the programmer will be alerted that the definitions are unsatisfiable. More information about errors is presented in Section 4.3.

2.3.1 Implicit Definitions

Some definitions which are represented explicitly in the internal representation are in fact implicitly applied in the programming environment. One such definition is the elem definition used to extract objects from groups. In the geometric syntax, there is no need to explicitly break objects out of groups, because they are already visible to the programmer in the data view.

We present such definitions alongside the others for completeness in Chapter 3. They preserve the uniformity of the internal representation, and are therefore valuable to implementers of the Construct programming environment and interpreter.

2.4 Applying Modifications

2.4.1 Object Modifications

The process of applying object modifications (Section 3.4.1) is quite similar to that of applying definitions. The programmer first selects the object to be modified, and then selects the modification from a tool palette.

2.4.2 Definition Modifications

The interactions for definitions modifications vary by modification. Each of them are outlined below:

not Modification

There are two circumstances when the not modification can be applied. If the programmer wishes to negate a definition which is currently being applied, then after selecting the definition from the appropriate tool palette, the programmer may select not from its tool palette. When the definition application is made, it will be negated.

If the programmer wishes to negate a definition which has already been applied to an object, the process is similar to the application of object modifications. The programmer first selects the definition, and then selects not from its tool palette.

handle Modification

handle requires an existing definition to be defined for the object ρ. The programmer selects this definition, then selects the handle modifier from its tool palette. The programmer must then select another definition to serve as the "failure" case of the handle. This definition is automatically applied to ρ. The programmer must specify reference objects for the definition as in normal definition application.

each Modification

The each modification must be specified during the initial definition application process. First, the programmer selects the object to be defined as normal. This object must have type set(τ). Then the programmer activates the each command from a tool palette, and then selects some definition δ_τ appropriate to an object of type τ. Finally, the reference object must be of type set(τ'), where τ' is the type of the reference object for δ_τ.

filter Modification

The interaction to apply a filter modification is identical to that of each.

2.5 Creating Local Definitions

Local definitions (see Section 3.5) allow the programmer to wrap up complex derivations of objects into a single definition without making that internal logic available to other programs. They are similar to anonymous functions in traditional text-based languages.

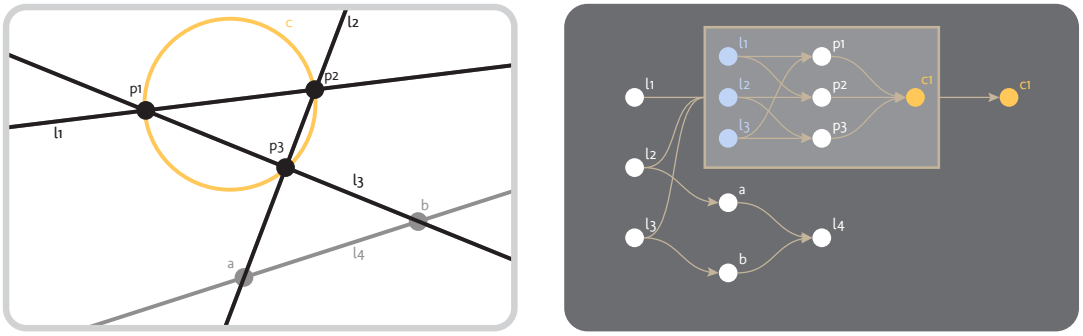

Figure 2.3: The appearance of a local definition in the program space (left) and graph (right)

Local definitions can be created from the beginning in a manner similar to the method of applying definitions. The programmer selects the object to define, then selects the local tool palette command. Unlike a predefined definition, the local definition has no references already defined.

The program space and graph will then appear similar to what is shown in Figure 2.3. The object which is to be defined will appear locally, marked as final. The programmer may then build up the objects and definitions used to define the final object. If the programmer references an object outside the local definition (distinguishable by fading of those objects), that object will automatically become a reference for the local definition and appear as an initial inside the local definition.

2.6 Definition, Not Manipulation

It is important to understand that while the Construct programming environment appears in many ways similar to a vector graphics editor, it does not operate by direct manipulation of objects. The nature of the objects in the program space is given by definition rules which establish relationships between the objects. Actually dragging or otherwise transforming the particular objects shown in the program space, while potentially supported for clarification purposes, will have no effect on the meaning of a Construct program.

The program space is presented to the programmer to provide an example of the data in the program at runtime, lifting the burden of mental simulation of the effects of the code. Construct is therefore not a programming by example [9] system, but rather *programming with examples* [5].

Chapter 3

The Structure of Programs

A program in Construct is a Definition (see Section 3.3) which follows a well-defined procedure to generate *final* objects from *initial* objects. Since a program is a Definition, it can used in other programs to abstract away higher-level operations used for generating more complex objects.

A program is a tuple:

$$(O, D, M_\rho, M_\delta)$$

O is the set of all objects (see Section 3.1) that are present in the program. D is a function $\rho \to \{\delta_1, \delta_2, \ldots\}$ which, given an object ρ, produces the set of all definitions (see Section 3.3) which describe ρ. M_ρ is a function $\rho \to \{\mu_1, \mu_2, \ldots\}$ which, given an object ρ, produces the set of all modifiers (See Section 3.4.1) which describe ρ. M_δ is a function $\delta \to \{\mu_1, \mu_2, \ldots\}$ which, given an object ρ, produces the set of all modifiers (See Section 3.4.2) which describe δ.

O and D may be combined to generate a directed acyclic graph G. Any valid topological sorting of G is a valid ordering of runtime execution for the program.

3.1 Objects

An object in Construct is an entity ρ which exists in 2d space. Objects may have position, orientation, magnitude, or any other characteristics which are appropriate based upon their Type (see Section 3.2).

Objects may be named by user-provided strings. These names will appear in the programming environment to disambiguate objects in complex algorithms. However, since the visual display of Construct disambiguates objects in a manner not possible in a text language, naming of objects is not necessary in simple programs. There are no restrictions on the format of object names (or even their reuse), as the environment does not rely on names to identify objects.

At any point in the execution of a Construct program, any object is a representative of the *valid parameter set*. This is a set which contains all parameter combinations which are valid based upon the Definitions (see Section 3.3) of the object.

3.2 Types

Construct is strongly typed and does not support parametric typing in any user-created definitions. Typing in Construct may also be considered explicit, by nature of the live data representation of the program state. It is not possible to compose an ill-typed Construct program through the environment, avoiding an often-confusing category of errors. The types available in Construct are listed below:

τ ::=	pt		A point in 2d space.
	ln		An infinite line in 2d space.
	circ		A circle in 2d space.
	ang		An angle in 2d space.
	dist		A distance in 2d space.
	set(τ)		A homogeneous collection of objects of type τ.
	grp($\tau_1, \ldots, \tau_n$)		A ordered collection of objects.

We can further distinguish the types of objects into three kinds. pt, ln and circ are *literals*, because they correspond to real objects. ang and dist are *measures*. They correspond to familiar geometric concepts, but don't have actual position in the real world, and so behave somewhat differently in the program space. Finally, set and grp are *aggregates*, which hold multiple other objects, allowing for complex data structures.

Additionally, some built-in Definitions (see Section 3.3) accept numerals as parameters. Numerals are shown here with a bar, for example $\bar{n}$ is some integer n. These numerals allow convenient specification of constant values. Programmer-created definitions can not accept numerals as inputs.

Because Construct is inherently connected to a graphical editing environment, each type listed above is associated with an iconic representation of its data. In the programming environment (see Section 2.1), these icons are displayed in tool palettes and denote the commands used to instantiate new objects of the corresponding type.

The icons shown above were chosen to relate to the programmer's existing vocabulary of geometric objects, and to be visually distinctive from each other. For example, we can consider the icons for set and grp: Because sets are homogeneous and unordered, the icon shows an unorganized cluster of identical triangles. By contrast, since groups are both heterogeneous and ordered, the icon shows a line, circle and triangle, arranged linearly.

3.3 Definitions

Definitions are rules that are used to describe an object based upon other objects. All of an object's definitions are evaluated at runtime, during Reconciliation (see Section 4.2).

Each Definition is a member of a Definition Class δ_τ, which is the class of all Definitions that describe objects of type τ. Each Definition's inputs and outputs must be a single object, though groups and sets may be passed. A Definition may be totally described in this notation (group syntax simplified for brevity):

$$\mathsf{defname}[\tau](\rho_1 : \tau_1; \rho_2 : \tau_2; \ldots)$$

If the Definition is listed as part of a Definition Class (as below), we omit the bracketed type information for brevity. Additionally, we do not specify the type in our syntax (see Chapter 5) because it can be derived from context.

Definition icons, like type icons, appear in the tool palettes of the programming interface. They also appear in the program graph as annotations to the edges in the graph, in order to quickly explain the relationships between objects in the program.

3.3.1 Generic Definitions

The following Definitions describe objects of any type τ. As noted above, Construct does not support parametric typing in user-created definitions. Only this small set of built-in definitions are applicable to all types of objects.

δ_τ	::=	null		This definition has no effect on this object. It may be used with the handle modifier (see Section 3.4.2).
		$\mathsf{id}(\rho : \tau)$		This object is identical to ρ.
		$\mathsf{of}(s : \mathsf{set}(\tau))$		This object is identical to a member of the set s. This definition is added implicitly in the programming environment when the user manipulates a set member.
		$\mathsf{elem}(\bar{m};\ g : \mathsf{group}(\tau_1, \ldots, \tau_n))$	[impl]	This object is the mth object in the group g. If $m > n$ or $\tau_m \neq \tau$, raise DefErr. This definition is added implicitly in the programming environment when the user manipulates an object inside a group.

The icons that represent these definitions were again chosen to align with symbols familiar to the programmer. The icon for null bas been chosen to resemble the common slashed-circle negation

symbol, though special attention was paid to create an icon which is visibly distinct from the similar icons for cross (see Section 3.3.3).

The id icon was chosen to resemble the mathematical equivalence operator ≡, rather than the more common equality symbol = because the latter symbol could be easily confused with the usual symbol for parallel lines, used by par (see Section 3.3.3).

The of symbol, as the extraction device for sets, has an icon very similar to the set type shown in Section 3.2. A single member of the set is distinguished in the of icon to call attention to its purpose.

One definition listed here, elem, does not have an iconic representation. This is because both in the program space and program graph, using an element of a tuple is most clearly presented as an implicit operation to the programmer. This definition exists only in the internals of the language and is never exposed to the programmer, and so does not require an iconic representation.

3.3.2 Definitions for pt Objects

δ_{pt} ::=	on(l : ln)		This point occurs somewhere along l.
	on(c : circ)		This point occurs somewhere along c.
	opp(p : pt; l : ln)		This point exists on the opposite side of l from p.
	inside(c : circ)		This point occurs inside of c.
	center(c : circ)		This point is at the center-point of c.
	to(p : pt; d : dist)		This point is at distance d from p.
	to(l : ln; d : dist)		This point is at distance d from the nearest location on l.
	to(c : circ; d : dist)		This point is at distance d from the nearest location on the circumference of circle c.

Here we see that the built-in definitions may be overloaded to apply to different combinations of arguments. When this is done, the icons for the definitions do not change, in order to minimize the number of icons that are shown in the tool palettes. Definitions which are overloaded always describe logically similar relations between objects.

For example, the on definition is overloaded because a point can be on a line or on a circle. To require the programmer to think about which object type the point is on is an excessive burden that does not align with how such relationships are ordinarily discussed in geometry. Though the internal calculations will be different, Construct exposes a logically consistent interface to the

programmer which allows such details to be handled transparently by the language.

The on definition is represented by a reticule icon to disambiguate it from several symbols which may appear similar. × or + are commonly used to indicate point objects in graphics editing programs; however, these symbols can potentially be confused with the symbols for intersection (see intr in Section 3.3.3). Therefore, the reticule symbol has been chosen to refer to the alignment of a point to another object. The circle component of the reticule has been reduced in size to likewise avoid confusion with the icon for the center definition.

To represent opp and inside, an curved arrow is shown pointing to the zone in which the point is intended to occur. The curvature of the arrow reinforces the sense of the line or circle as a boundary applied to the object (which is then being "leapt over"). Additionally, the icon avoids a straight arrow to avoid the assumption of a distance or direction relation to the referenced objects.

The center definition does make use of a traditional locus symbol, since there is no possibility of confusion with any of the other symbols presented to the user, due to the surrounding circle.

There were several challenges to the design of the to definition icon. 1) to will be a highly overloaded definition, having to accommodate all of the possible combinations of input and output objects; 2) the icon should be visually distinct from the icon representing the dist type; and 3) the icon should not suggest directionality because distances in Construct are scalar. Therefore, the to icon shows a point and a line, with a spanning line segment that represents the distance between. The gaps between the objects and the distance are similar to gaps left in traditional architectural dimensioning notation.

3.3.3 Definitions for ln Objects

δ_{ln} ::=	thru(p : pt)		This line passes through p.
	intr(l : ln)		This line intersects l at some location.
	par(l : ln)		This line is parallel to l.
	perp(l : ln)		This line is perpendicular to l.
	skew(a : ang; l : ln)		This line is at angle a to line l.
	tan(c : circ)		This line has a point of tangency to c at some location.
	cross(c : circ)		This line intersects c at two locations.
	to(p : pt; d : dist)		This line is at distance d from point p at it closest location.
	to(l : ln; d : dist)		This line is at distance d to line l at their closest points. In two dimensions, this implies par(l).
	to(c : circ; d : dist)		This line is at distance d from the nearest location on the circumference of circle c.

The definitions for ln objects continue to resemble symbols used in pen-and-paper geometry to conform to user expectations. par, perp, and tan should be immediately recognizable to users as the geometric relationships they represent.

The intr icon was designed so that the intersection of its lines was not at right angles to minimize confusion with perp. Similarly, skew features a slight extension of its lines past the point of intersection to distinguish it from the sharp-cornered icon for the ang type. The icon representing cross was likewise designed to be distinct from the icon for null introduced above, by extending the diagonal line outside the bounds of the circle and orienting it along an opposing diagonal.

3.3.4 Definitions for circ Objects

δ_{circ} ::=	thru(p : pt)		This circle passes through p.
	about(p : circ)		This circle is centered on p.
	tan(l : ln)		This circle has a point of tangency to l at some location.
	tan(c : circ)		This circle has a point of tangency to c at some location.
	cross(l : ln)		This circle intersects l at two locations.
	to(p : pt; d : dist)		This circle's circumference is at distance d from point p at it closest location.
	to(l : ln; d : dist)		This circle's circumference is at distance d to line l at their closest points.
	to(c : circ; d : dist)		This circle's circumference is at distance d from the nearest location on the circumference of circle c.

Here we can see that many of the icons are reused from complimentary definitions for pt and ln objects. Notice that the icons from thru and tan are not altered to display circular arcs instead of lines. The icons remain the same so as not to confuse users by changing the interface in response to other actions. A programmer should be able to depend on the consistency of the buttons shown in the tool palette throughout the process of development.

3.3.5 Definitions for ang Objects

δ_{ang} ::=	join(a : ang; b : ang)		This angle's sweep is equal to the sum of the sweeps of a and b.
	split(a : ang; $\bar{n}$)		This angle's sweep is equal to $\frac{1}{n}$ times the sweep of a. If $n = 0$, raise DefErr.
	sweep($\bar{n}$)		The sweep of this angle is equal to n (in degrees).
	between(k : ln; l : ln; p : pt)		This angle's sweep is the sweep between the lines k and l which contains p.

A particular challenge in designing the icons for Construct was in the design of the definitions for ang objects. The join and split definitions have a very similar appearance in pen-and-paper geometry, often disambiguated only by an understanding of the procedure that is being performed. To clarify the Construct icons, the join icon is presented as the combination of two clearly distinct angles, having different breadth and also marked with arcs that do not align. In contrast, the icon representing split shows two angles of identical size, spanned by a single continuous arc. In this way, the icon for split should evoke the image of an angle bisector, familiar from traditional geometry, though the Construct definition is more powerful than that technique.

The icon representing sweep shows the mathematical variable n beneath a degree sign, clarifying the units of the operation and providing a connection to a well-understood geometric concept. When this definition is applied to an object, the numeric argument is presented in place of n, mimicking usual geometric notation.

3.3.6 Definitions for dist Objects

δ_{dist} ::=	$\mathsf{sum}(d : \mathsf{dist};\ t : \mathsf{dist})$		This distance is equal to the sum of the lengths of d and t.
	$\mathsf{div}(d : \mathsf{dist};\ \bar{n})$		This distance is equal to $\frac{1}{n}$ times the length of d. If $n = 0$, raise DefErr.
	$\mathsf{length}(\bar{n})$		This distance is equal to n.
	$\mathsf{span}(p : \mathsf{pt};\ q : \mathsf{pt})$		This distance is equal to the distance from p to q.
	$\mathsf{span}(p : \mathsf{pt};\ l : \mathsf{ln})$		This distance is equal to the shortest distance from p to l.
	$\mathsf{span}(p : \mathsf{pt};\ c : \mathsf{circ})$		This distance is equal to the shortest distance from p to c.
	$\mathsf{span}(k : \mathsf{ln};\ l : \mathsf{ln})$		This distance is equal to the shortest distance from k to l.
	$\mathsf{span}(l : \mathsf{ln};\ c : \mathsf{circ})$		This distance is equal to the shortest distance from l to c.
	$\mathsf{span}(b : \mathsf{circ};\ c : \mathsf{circ})$		This distance is equal to the shortest distance from b to c.

As with ang definitions, the definitions for dist objects presented ambiguities to resolve. The sum and div definitions are presented in a manner similar to the approach above. sum shows two clearly different distances, offset from each other, whereas div presents a binary division into two

identical distances. As with split, div provides greater flexibility than the analogous pen-and-paper geometry technique, which can only perform binary division.

length is represented in the same manner as sweep above. When applied, the numeric argument is shown in the definition symbol.

3.3.7 Definitions for set(τ) Objects

δ_{set}	::=	empty		This set has no members.
		size($\bar{n}$)		The number of members of this set.
		include($\rho : \tau; s : \mathsf{set}(\tau)$)		This set contains all members of the set s and additionally contains ρ.
		exclude($\rho : \tau; s : \mathsf{set}(\tau)$)		This set contains all members of the set s except ρ. If ρ is not a member of s, raise DefErr.

The definitions of set objects pose a unique challenge to represent in graphical form. Sets, unlike the other object types which have been previously addressed, are not clearly represented in a visual data view. Similarly, the operations that are commonly performed on sets do not have a literal representation in pen-and-paper geometry. Set-based operations are often undertaken solely in the mind of the geometer during the algorithm, but must be represented in Construct programs.

Therefore, the set icons borrow from non-geometric mathematics. empty uses the common representation of an empty set as two curly braces enclosing no elements. While this does not relate directly to geometric views of a set, the icon does avoid an ambiguity with the icons for null and cross, which resemble the alternative slashed-zero representation of an empty set.

size is represented as finite set cardinality notation in discrete mathematics, and undergoes the same transformations as the other numeric definitions when applied to a set object.

Solid circles were selected as an abstract representation of set objects for the include and exclude definitions. The include icon evokes the addition of a smaller, like element to the set, while the void in the exclude icon shows the removal of an element.

Of the categories of icons presented in this document, the icons for set definitions have the least visual clarity. An area of future work will be to reconcile the requirements for set icons to produce symbols which can be clearly recognized by users.

3.3.8 Definitions for grp($\tau_1, \ldots, \tau_n$) Objects

δ_{grp}	::=	collect($\rho_1 : \tau_1, \ldots, \rho_n : \tau_n$)	[impl]	This group contains $\rho_1, \ldots, \rho_n$.

Like the elem definition (see Section 3.3.1), collect need not be explicitly available in the programming environment. Groups are constructed by the group typing tool and visually marked as

combined objects in the program space and program graph. This definition is used internally by the implementation.

3.4 Modifiers

Modifiers are higher-level constructs than definitions which further specify program behavior. They can apply to either objects or definitions, and alter the meaning of each.

Modifiers mark objects and definitions in ways that do not have a clear visual representation in traditional geometry. Therefore, the icons used in Construct are more symbolic in nature than those designed for definitions.

3.4.1 Modifiers for Objects

μ_ρ	::=	initial		ρ is given as an input to the program.
		final		ρ is the result of the program.
		unique		The Definitions which specify ρ must resolve to exactly one possible set of parameters. If ρ is not uniquely defined, raise DefErr.

The icons designed for initial and final mirror the presentation of those object in the program graph. While it is possible for definitions to apply to or reference either, in most Construct programs definitions will only flow from initial objects and toward final objects.

Uniqueness of definition is a constraint that may be useful to programmers to check programs for logical or input data errors. However, it too has no analog in traditional geometry. Therefore, the iconic representation is a star, simply denoting that the object is "special" to the program.

3.4.2 Modifiers for Definitions

μ_{δ_τ}	::= not		The negation of this Definition.
	handle(δ'_τ)		If this Definition raises a runtime error, instead apply δ'_τ.
	each		Applies this Definition over all members of a set, specifying a new Definition in the class $\delta_{\mathsf{set}(\tau)}$. If DefErr is raised at any point, this whole definition raises DefErr.
	filter		Applies this Definition over all members of a set. If the definition does not raise an error, include its *final* object in the new set. Otherwise, skip that object without raising DefErr.

The icon representing not was selected both to correspond with the negation symbol in some existing programming languages, but also to remind the programmer that the definition is behaving in a manner opposite normal expectations. When applied to a definition, the not icon prefixes the definition's usual symbol. The bang symbol ! was chosen rather than the logical negation symbol ¬ because the logic symbol is entirely unfamiliar to non-mathematicians, and is also reminiscent of perpendicular lines, which could cause user confusion.

The handle icon resembles the wavy arrow which is used to represent handling in the program graph. It suggests a dependency between elements, but that this dependency is looser than usual.

each and filter use representation similar to that used with sets to indicate objects that are considered or left out in their operations. However, these icons suggest iteration, which is not a part of the semantics of the modifiers (all computations are strictly independent). A less-sequential design of these icons would be a valuable future improvement to the presentation of Construct modifiers.

3.5 Local

λ	::= local(G)		Produces a local definition from the program subgraph G. This can be used in conjunction with the handle, each and filter modifiers to apply more complex behaviors.

The design of the local icon mirrors the appearance of the local syntax in the program graph, as shown previously in Figure 2.3. By boxing the computation that is done inside the local definition, this representation reinforces the nature of the definition as separate from the main program.

Chapter 4

Execution

4.1 Runtime Errors

ϵ	::=	DefErr	A set of Definitions (see Section 3.3) could not be Reconciled (see Section 4.2) to a valid object.

4.2 Reconciliation

When an object in the Evaluation DAG becomes definable (when all of its immediate prerequisites are defined), the Construct runtime begins a process of Reconciliation. During Reconciliation the runtime attempts to find a state for the object which satisfies all of the Definitions applied to it. It selects a candidate object from the set of all objects which satisfy the constraints placed upon it.

4.3 Failure

During runtime, it is possible for Reconciliation to fail by an object possessing conflicting definitions.

This may initially occur as a result of choices made during the reconciliation of ancestor objects in the program. The runtime will first try to resolve the failure by redefining these ancestors in a way that will produce a valid result. If this is unsuccessful, the runtime declares DefErr.

A runtime error is then propagated back along the chain of ancestor definitions from the definition which triggered it. If the runtime encounters a definition which has a handle modifier attached to it, it will stop this propagation and attempt to redefine the program using the alternate definition. This definition itself may fail, which repeats this process on the changed program graph.

If the runtime error propagates all the way to to the initial objects at the top level of the program, the program has failed. The user should be alerted to the failure, and if the program is being evaluated in the programming environment, specific debugging information should be made available.

Chapter 5

How to Program in Construct

In this chapter, we discuss points relevant to prospective programmers in Construct. Common programming idioms are shown, as well as some examples of functionality that can be built using the tools we have described previously.

5.1 Idioms

In order to develop programs analogous to those commonly created using modern programming languages, programmers will make use of *idioms* in Construct. Here an idiom refers to a commonly-reused structure in programs that encapsulates a single piece of program logic. Below we present an example idiom developed within Construct; idioms for other behaviors in modern programming languages can be developed from the operations provided in Construct.

5.1.1 If Then Else

The idiom for *if-then-else*, shown in Figure 5.1, is distinguishable primarily in the graph view of a Construct program. We use a set object to represent the condition of the *if*: a set containing some elements represents *true*, whereas an empty set represents *false*.

With this definition of *true* and *false*, we use two local definitions to encapsulate the definitions occurring in the *then* and *else* branches of computation. In the *then* branch, we try to take an element out of the set. If the runtime is able to do this, then the condition must have been true, and the rest of the *then* branch definitions can be computed, arriving at some result r.

If the set is empty, then extracting an element will raise DefErr. This is picked up by the handle, shown in red, which causes the *else* branch computation to occur, again arriving at some result r. Then the two branches are unified onto the ultimate result object.

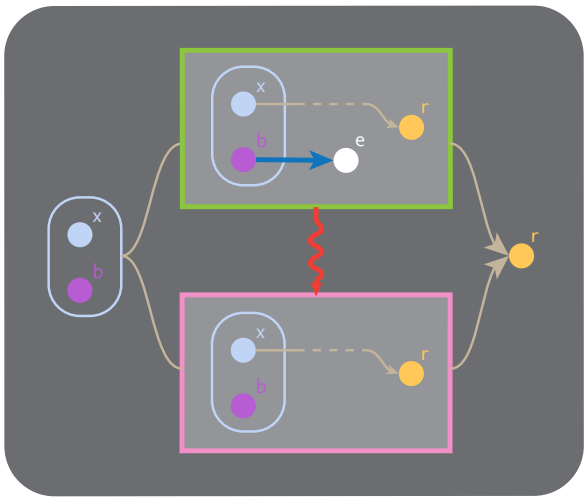

Figure 5.1: An idiom showing if-then-else behavior through the use of a set and handle.

5.2 Example Program

Below is shown an example program that can be written using the features of Construct described earlier in this document. The example begins with a view of the program in its final form, showing both the program space with the concluding state of the program and the program graph of the definition dependencies of the whole program.

Though the program graph is drawn left to right to show the dependencies, it is important to remember that Construct programs can be composed in any order. The programmer may choose to begin at the result of the program, and filling in the dependencies until reaching a satisfying construction. Alternatively, the program may be built from both ends, working the initial and final values together by creating intermediate objects.

5.2.1 Triangle Area

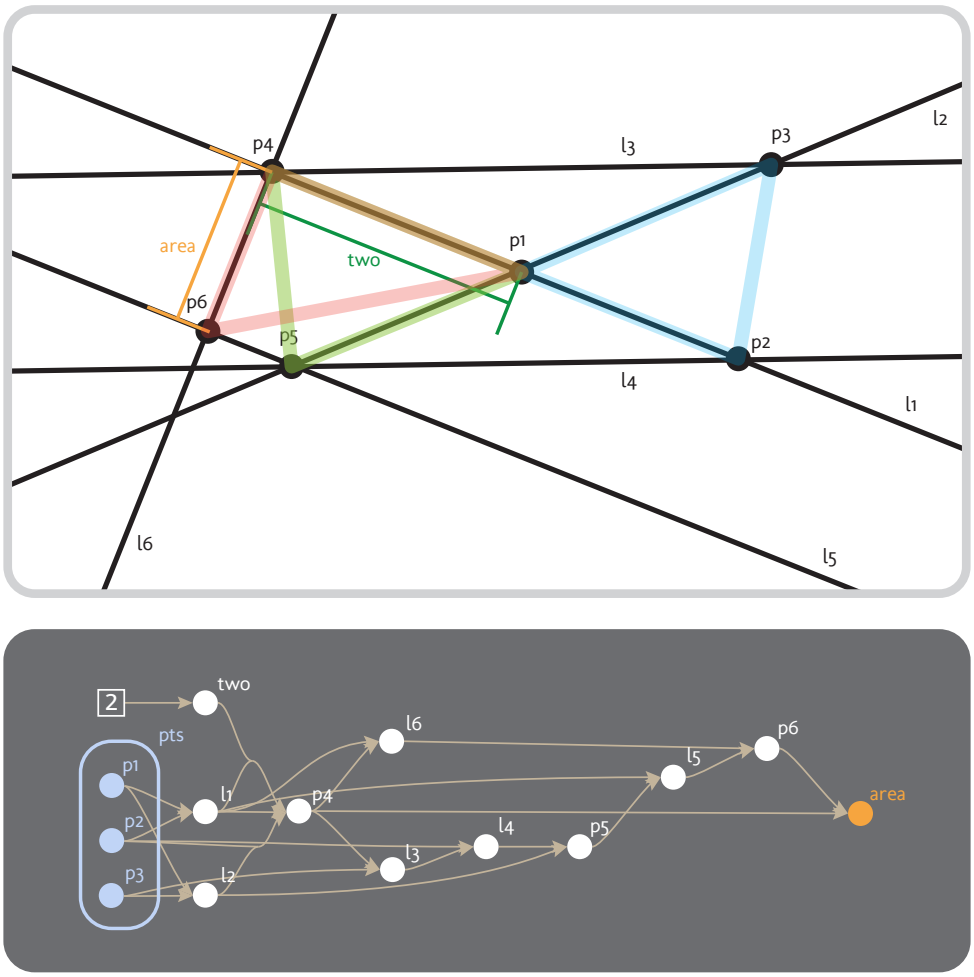

Figure 5.2: A Construct program which finds the area of a triangle formed from three input points using elementary geometric reasoning.

Figure 5.2 shows a program which calculates the area of the triangle in both the program space and program graph views. In the program graph, the three triangles which are constructed during the program's computation are highlighted successively in blue, green and red. This is not a feature of the Construct programming system, though it may be a useful formatting annotation in the programming environment.

In brief, this program works by constructing a triangle (p_1, p_4, p_5) which has the same area as the original triangle, where side $\overline{p_1 p_4}$ has length 2. Then a triangle (p_1, p_4, p_6) is constructed which is a right triangle. Since one leg of the right triangle is length 2, applying the standard formula for the area of a triangle reveals that the length of the other leg is equal to the area.

Chapter 6

Conclusions

6.1 Future Work

At this time, Construct exists as a general specification of a programming system. Future work on this system may take many directions, some of which are described below.

6.1.1 Runtime Implementation

This document only describes the behavior of Construct programs. It will be necessary to produce a working interpreter or compiler than can enable these programs to be actually written and tested. This runtime system will require a constraint solver that can efficiently step the program forward so that Construct is a viable alternative to traditional programming languages. Recent research into evaluation of constrained geometric systems [1, 3, 7] may provide insight into the development of a performant interpreter.

6.1.2 User Interface

In order to assess the usability of the interface to Construct, future work should explore a prototype of the programming environment described in Chapter 2. This prototype can then be tested with the target audience of visual professionals to gauge its effectiveness and gain user feedback on the design decisions made in the system.

6.1.3 Cut and Paste

An interface challenge to be explored in future work is the functionality of cut and paste. In textual programming languages, cut and paste can be performed independently of the surrounding text because there are no explicit linkages. Because Construct programs are presented to the programmer as directed graphs, performing cut and paste requires some trade-off between preserving edges (representing definitions), and maintaining predictability of the command to the programmer.

There has been some work on interfaces for cut and paste in graphs [4]. Expansion of this work and adaptations to the particular format of Construct's program graph will be a valuable addition to the usefulness of the programming environment. A corresponding cut and paste without the program space will also need to be developed, which is aware of the dependencies established in the program.

6.1.4 Higher Dimensions

Presently, Construct has been designed to facilitate geometric procedures solely in a two-dimensional plane. This is a significant limitation on the problems which are easily represented by the system. While approaches from projective geometry may be used to simulate working in higher dimensions, such approaches require the programmer to mentally simulate the true operation of the program, which falls short of Goal 1 (see Section 1.1). Extension of the design of Construct to three dimensions in future work could alleviate this problem.

6.1.5 Datatypes

Programmers will likely desire to shorthand complex datatypes composed of sets and groups into single entities. For example, programmers may desire to work with finite-length line segments rather than infinite lines. Currently, segments must be composed from the existing types in Construct, which does not create a clean representation of the programmer's mental model. Future work may explore adding the facility to Construct for arbitrary higher-complexity datatypes to be reduced to simpler programmer-created type definitions, possibly with the assistance of a module system.

6.2 Discussion

In Section 1.1 we laid out four goals to help programmers to use Construct. Each of these principles have shaped the design of this programming system.

6.2.1 Make clear the program state

Construct programs are shown to the programmer in an environment (Section 2.1) which displays both a graph of the evaluation of the program and a representative state of its objects. Though the runtime state is shown during composition, some elements of Construct programs are still difficult to represent in a geometric view. `set` objects and the effects of branching from `handle` modifiers do not have clear geometric analogues, and so may not be clear to the users of Construct.

In ongoing work on the Construct user interface and runtime, we hope to evolve the presentation of these elements so that users are able to clearly understand these elements.

6.2.2 Work in the native representation of geometric relationships

Chapter 3 describes the many features in Construct which mimic their traditional pen-and-paper geometry counterparts. The objects available in Construct have familiar analogs in geometry and their definitions match the behavior of traditional geometric objects. Additionally, the infix ordering of definition application mimics the description of geometric relationships in natural language.

However, some elements of Construct are not as familiar to traditional geometry. The handle modifier described in Section 3.4.2 does not have a meaningful visual representation. It is nonetheless representative of the need to perform different constructions based on the nature of the objects provided as inputs to a problem. In traditional geometry, the logic of this operation is not often drawn, and so devising a clear visual presentation should be a continuing area of work on Construct.

6.2.3 Specify relationships without inference

Construct is designed to be programmed entirely by specifying unambiguous rules. This methodology in designing the language means that we never rely on machine inference, so the evaluation of the programs is always specified directly by the user.

However, Construct also permits specifications of geometric systems which are ambiguous. Definitions may be used in such a way that objects are not uniquely defined, which could make it difficult for programmers to reason about some programs. Future user testing will allow us to understand where this may cause problems for users, and modify the language and/or programming environment to mitigate this confusion.

6.2.4 Run a full (Turing-complete) range of programs

Construct aims to provide its users a fully-powerful programming system. The capability for branching is provided by the handle modifier, and demonstrated in Section 5.1.1. Since all programs written in Construct are definitions, the user definition drawers (shown in Section 2.1) allow the programmer to recursively use other definitions in programs. The existence of these two features strongly suggests the Turing-completeness of this programming system.

The Construct programming environment has the potential to prevent the development of nonterminating programs, however. Because the program space shows the runtime state of Construct programs, a nonterminating program would cause the environment to enter an infinite loop. The ability to represent nonterminating programs in the environment is an important design consideration for future work on programming interface for Construct.

6.3 Contribution

Construct introduces a new way to program geometric computations. By offering simultaneously information about the evaluation dependencies and runtime state of a program, Construct provides programmers with more information about their programs during composition.

Additionally, Construct presents geometric algorithms natively through visual representations,

including the higher-level logic necessary to explain recursive and branching processes. This alternative representation of computations should enable programmers to think about problems in ways not encouraged by traditional languages, possibly helping to discover new solutions and approaches.

Bibliography

[1] Alberti, M. A., Evi, P., and Marini, D. Modelling constrained geometric objects with OBJSA nets. In *Concurrent object-oriented programming and petri nets*. Springer Berlin Heidelberg, 2001, pp. 319–337.

[2] DeRose, T. D. A coordinate-free approach to geometric programming. In *Theory and practice of geometric modeling*. Springer Berlin Heidelberg, 1989, pp. 291–305.

[3] Freixas, M., Joan-Arinyo, R., and Soto-Riera, A. A constraint-based dynamic geometry system. *Computer-Aided Design 42*, 2 (2010), 151–161.

[4] Ibrahim, B. Optimizing cut-and-paste on directed graphs, with a user-controlled edge reconstruction strategy. In *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on,* (1998), pp. 90–91.

[5] Myers, B. A. Visual programming, programming by example, and program visualization: a taxonomy. *ACM SIGCHI Bulletin 17*, 4 (1986), 59–66.

[6] Nelson, G. Juno, a constraint-based graphics system. *ACM SIGGRAPH Computer Graphics 19*, 3 (1985), 235–243.

[7] Pion, S., and Fabri, A. A generic lazy evaluation scheme for exact geometric computations. arXiv preprint cs/060863, 2006.

[8] Reiser, R. H., Costa, A. C. R., and Dimuro, G. P. Programming in the geometric machine. *Frontiers in Artificial Intelligence and Its Applications, Amsterdam: IOS Press 101* (2003), 95–102.

[9] Smith, D. C. *Pygmalion: A creative programming environment.* PhD thesis, Stanford University, 1975.

[10] Sutherland, I. E. Sketchpad: A man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, spring joint computer conference* (1963), pp. 329–346.

[11] Victor, B. The future of programming. `http://vimeo.com/71278954`, July 2013. [Online; accessed 11-April-2014].

[12] Victor, B. Media for thinking the unthinkable. `http://vimeo.com/67076984`, April 2013. [Online; accessed 11-April-2014].

[13] Victor, B. Stop drawing dead fish. http://vimeo.com/64895205, April 2013. [Online; accessed 11-April-2014].