Enhancing Functional PostScript by Constraint Techniques

Markus Holmberg

February 14, 2004

Abstract

Graphics systems in abstraction layers closer to the rendering layer tend to require the properties of individual graphics objects to be defined explicitly. Allowing these properties to be defined declaratively at a higher level provides a number of benefits related to the higher abstraction level. Employing a technique of declaratively describing acceptable values of a property but leaving the derivation of the actual value to an underlying system is employing a *constraint technique*.

One such approach is to allow properties to be defined in terms of mathematical expressions, which may include other properties to create dependencies. These definitions result in a set of equations describing relations of properties of graphics objects involved in a picture. Given sufficiently informative property definitions it is possible to infer explicit values from the equations, which then can be passed to a lower-level graphics layer.

A system allowing properties to be defined in relation to each other has been created to enhance the Functional PostScript graphics system. This report presents its design and use, with the additional intent to show how constraint techniques can be leveraged in graphics systems.

Acknowledgements

I would like to thank Frank Drewes, my advisor, for inspiration, guidance and support throughout the project.

Contents

List of Figures

1 Introduction

An approach to computation where *what* to compute is specified differs from an approach where *how* to compute something is specified. The former is sometimes called a *declarative* approach, while the latter is called an *imperative* approach. The former has a number of advantages, such as often being the easier approach for the user, but also disadvantages such as intractable computational complexity in certain circumstances. Nevertheless, the former is a valuable addition to the toolbox of approaches.

An approach to specifying *what* to compute, is that of declaring *constraints* on the result. Constraints limit the range of what is an acceptable result. This approach is especially usable in the context of computing graphics, which is the subject of this thesis.

1.1 Background

Functional PostScript [12] is a system for producing device- and resolution-independent graphics from Scheme programs, by providing an abstraction of PostScript's [4] base imaging operators in the Scheme [5] language. This combination is advantageous, as it combines the powerful PostScript graphics primitives with the expressiveness of the Scheme language.

PostScript is a page description language, that is, a language for describing the contents of a printed page in a higher level and more compactly than the actual output bitmap. The imaging model of PostScript considers an image to be built up by placing ink on selected areas of a page. This is done by placing out text and graphical shapes on the page using the PostScript graphics operators, with positions on the page described with x and y pairs in a coordinate system imposed on the page. This means that each object is positioned in absolute values relative to the coordinate system origin, independent of the positioning of other, possibly conceptually related, objects. It is likely that in a picture composed of multiple objects, the positioning of the objects are thought of in terms relative to each other. It is therefore of interest to provide facilities to express such relationships.

Furthermore, geometric figures (shape and position) can often be defined uniquely using several different combinations of data. For example, a rectangle can be uniquely defined using different combinations of the locations of its corner points, width, height and so on. The PostScript rectangle graphics operator requires a specific combination to be used to define a rectangle (lower left point, width and height). From a user's point of view, this is an arbitrary limitation. Therefore, it is of interest to free the user from this limitation.

Both previously mentioned problems, relative positioning and multiple ways of uniquely defining figures, boil down to stating relationships between properties of objects in declarative form. More generally, the problem of inferring values for properties (positions, sizes etc) from a set of declarative statements about the relations between the properties of the objects can be viewed as a *constraint satisfaction problem*. A constraint satisfaction problem is a general problem in which the goal is to find values for a set of variables that will satisfy a given set of constraints.

1.2 Objective

The objective is to investigate how constraint techniques can be employed in a system producing graphics using Functional PostScript. This involves exploring a number of design issues, as well as the implementation of a prototypical software system.

The imagined target use of the system is producing technical illustrations, schematic figures, diagrams and other figures with a high degree of regularity. To start with, the types of constraints that are suitable for use with the targeted figure types need to be found out. These then need to be considered in regard to the existence of solving algorithms, their efficiency and the implementation effort required. After a design has been decided on, a prototypical implementation can take place.

Existing systems that already target production of figures with high regularity using constraint techniques do exist. However, they are often inventing their own languages in which constraints are fundamentally integrated. From that point of view, this project is different. In the same way that Functional PostScript provides access to the PostScript base imaging operators as a Scheme component, it is going to provide a constraint system that can function as a component orthogonal to the graphics system provided by Functional PostScript. That is, the focus is solely on the constraint system as a general component.

In summary, the area of constraint satisfaction problems as used in the field of graphics will be researched, and the acquired knowledge then applied to the case of designing a constraint system for use with Functional PostScript.

1.3 Terminology

Certain frequently occurring terms bear different meanings depending on the context, and are worth clarifying to avoid confusion. A *system* can refer to either a collection of software, or a collection of items, such as equations. In the context of this work, the term *constraint system* refers to a software system, concepts and software routines included, employing constraint techniques. Due to the common use of the term *equation system* to refer to a collection of equations in combination with equations being a form of constraint, the term could easily be misinterpreted, i.e., the reader may erroneously assume that it refers to a collection of constraints. When a collection of constraints is referred to, the term *system of constraints* is used instead.

1.4 Functional PostScript

Functional PostScript is a portable system for doing device- and resolution-independent graphics from Scheme programs. Since Functional PostScript is the intended target graphics system of the work reported here, the ideas behind Functional PostScript, a little on how it works, and a couple of code examples for it will be presented below.

1.4.1 Ideas

PostScript is a good target for a system seeking a model for describing 2D graphics: renderers for it are ubiquitous, it provides simple but powerful graphics

primitives, and it comes with a rich font handling system. A fundamental observation regarding PostScript that must be made to appreciate the idea behind Functional PostScript is that the PostScript page description language includes, in addition to theoretically being a general-purpose programming language, a model of how pictures are described using a set of graphics primitives.

The PostScript *graphics operators* are accessed through the PostScript *programming language*. The syntax of the PostScript programming language most closely resembles that of the programming language Forth. It also shares several other traits with Forth, perhaps most notably the postfix notation that follows the stack-orientation of the language. The PostScript graphics operators are excellent for expressing pictures, but cumbersome to use from the PostScript programming language.

What Functional PostScript does is that it takes the graphics operators provided by PostScript and embeds them in a Scheme engine, instead of a Forth-like PostScript engine, in effect replacing the PostScript programming language with the Scheme programming language. This enables pictures to be expressed in terms of PostScript's graphics operators, while taking full advantage of the expressiveness provided by the Scheme language. That is, Functional PostScript provides a different vehicle for expressing pictures in terms of PostScript graphics operators. This places Functional PostScript in the layer just above the PostScript interpreter, as a medium for communicating expressions in PostScript graphics operators.

1.4.2 Producing Graphics

The "Functional" in Functional PostScript refers not only to the use of Scheme but also to the side-effectless style of operation when working with graphics objects. Complex pictures are built up by combining smaller and simpler pictures, which can in turn be composed of smaller and simpler pictures themselves, in a functional style. Ultimately, pictures consist of what can be produced with the PostScript graphics primitives. This includes lines, arcs and curves, with the addition of glyphs for text rendering.

The process of producing graphics using Functional PostScript involves executing a Scheme program using the Functional PostScript library to produce PostScript output, that in turn is passed to a PostScript interpreter for rendering. This process is depicted in Figure 1.

To interface PostScript renderers, Functional PostScript uses a *channel* abstraction. Channels interpret picture expressions, usually into an external form. The current release of Functional PostScript comes with a default channel implementation that outputs text streams representing PostScript Language Document Structuring Conventions-compliant PostScript programs, but other channels such as Display PostScript for rendering directly to a screen are possible too.

1.4.3 Code Examples

To show what programs in PostScript and Functional PostScript respectively may look like, examples of the popular "Hello World" program will be given. The examples are not intended to convey any specific virtues of the two languages; they are included to provide a sample of what programs in them may

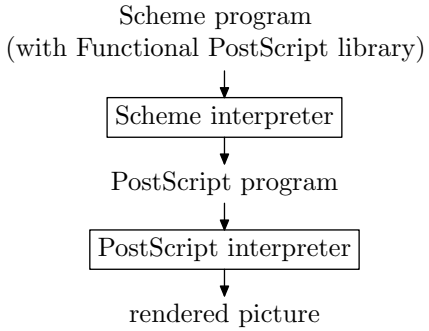

Figure 1: Producing graphics with Functional PostScript

look like.

```
/Courier findfont 12 scalefont setfont
0 0 moveto
100 100 translate
(Hello World) show
showpage
```

This is the PostScript version of "Hello World." The first line sets the current font, which is required for rendering text. The second line establishes a current point, which is required for subsequent operations that operate relative to the current point, at the origin. The third line modifies the current transformation matrix to map the positioning of the text, that shortly will be placed out, a bit further into the page. The fourth line creates and strokes the path representing the rendition of the string "Hello World" in the current font at the current position. The fifth and last line says that the current page is ready for rendering.

```
(stroke
 (translate 100 100
  (string->glyphpath (font "Courier" 12) "Hello World")))
```

This is the approximately corresponding Functional PostScript program. It strokes a path translated into a position further into the page, with the path being the path constructed from combining glyphpaths from the selected font corresponding to the characters of the the string "Hello World."

A slightly more complex Functional PostScript example follows below. It defines a procedure `snowflake`, with helper procedures, for generating Koch snowflake fractals.[1]

```
;; Snowflake in 'n' iterations
(define (snowflake n)
  (let ((side (translate 0 (/ (sqrt 3) 6)
                (snowflake-side n))))
    ;; Start with equilateral triangle
```

[1] Helge von Koch was a Swedish mathematician who started the investigation of such fractal curves. [7]

```
    (compose
     (rotate (* 0/3 2pi) side)
     (rotate (* 1/3 2pi) side)
     (rotate (* 2/3 2pi) side))))

;; Side of snowflake in 'n' iterations
(define (snowflake-side n)
  (if (= n 0)
      ;; Base case, draw line
      (line (pt -1/2 0) (pt 1/2 0))
      ;; Recursive case, remove middle and descend to next level
      (let ((side (scale 1/3 1/3 (snowflake-side (- n 1)))))
        (compose
         (translate -1/3 0 side)
         (rotate-around -1/6 0 (*  1/3 pi) side)
         (rotate-around  1/6 0 (* -1/3 pi) side)
         (translate  1/3 0 side)))))

;; Rotate path around point (instead of around origin)
(define (rotate-around x y angle path)
  (translate x y
   (rotate angle
    (translate (- x) (- y) path))))
```

The `snowflake` procedure takes an argument `n` indicating the number of iterations to use in generating the fractal. It then takes help of the `snowflake-side` procedure, that takes the same argument `n` of the same meaning, to generate the three similarly shaped parts of the fractal. Examples of the output are shown in Figure 2. The left-hand side of the figure shows how a side of the initial equilateral evolves through iterations 0 to 3. This output is produced by invoking `snowflake-side` with `n` equal to 0 through 3. The right-hand side shows the complete fractal resulting from invoking `snowflake` with `n` equal to 3.

The fractal is most easily described in a recursive manner, which is a style of building up pictures where Functional PostScript excels. Starting with an equilateral triangle, each side is divided into three parts with the middle part removed and replaced with two new lines of the same length, which in turn are processed equivalently, recursively the desired number of times. The program requires only one of the graphics element primitives, the line, but uses all three transformation operators, i.e., `translate`, `scale` and `rotate`.

2 Related Work

There exists some amount of previous work exploring the use of constraints in graphics and user interface applications, which has provided inspiration and experience to draw from.

The primary sources of inspiration are the figure drawing systems METAPOST and IDEAL. Both have designed their own languages for expressing pictures, mainly aimed at producing technical illustrations, and are operated in batch mode. In addition to batch mode-operated systems, there also exist interactively operated systems that utilize constraints. These are usually either

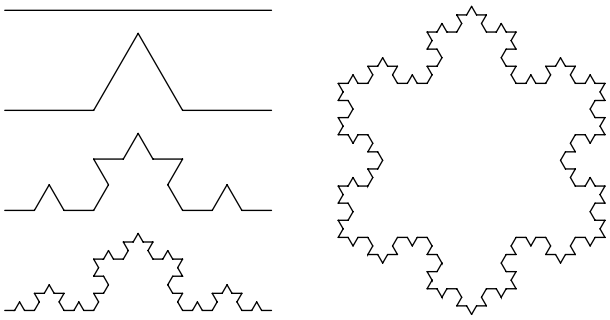

Figure 2: Output of Functional PostScript example

figure drawing applications, such as Juno, or toolkits for use in user interface specification, such as Cassowary.

2.1 METAPOST

METAFONT [6] is a system for designing fonts that was created by Donald E. Knuth as part of the TEX Project. METAFONT targets the creation of bitmap fonts. It uses a mathematical approach to font design, where each character is described in terms of mathematical curves and lines. This gives it the advantage of being able to produce entire font families from a single parameterizable source, whereas other approaches to bitmap font design require each different size of the same typeface to be crafted by hand.

Even though METAFONT is targeted at font design, it is perfectly suitable for other tasks, such as designing logos, and figures in general. While the figure descriptions are resolution-independent, the output is not, which hence limits its usage. Therefore, recognizing the capabilities of the METAFONT system, the figure drawing system METAPOST [3] was derived from it. METAPOST is a modified version of METAFONT which intercepts the rendering process just before rasterization, and instead of bitmaps outputs figures in the resolution-independent PostScript language.

Each figure (or in METAFONT, each character) is described by a program in the METAPOST language, which is a mix of imperative and declarative statements. Declarative statements include equations containing variables that represent points and sizes etc, that in turn are used in imperative drawing statements. In effect, the substitution technique for solving systems of linear equations is performed in parallel with equations being read in to solve the system of linear equations that is made up by the equations of the program. However, nonlinear equations are allowed as long as the substitutions based on the equations leading up to the nonlinear equation make it linear. This was deemed to be sufficiently

expressive for the targeted use.

2.2 IDEAL

IDEAL [11] is a block-structured declarative programming language for expressing pictures, intended for use in conjunction with text-formatting systems.

The central concept of "boxes" in IDEAL provides an abstraction facility. Boxes include a system of constraints—equations in points—that declare their relative position in a picture. They are additionally associated with drawing commands, which are executed in positions relative to the points of the box. Locations and properties of a box, such as the top left, top right, bottom left, bottom right corners, width and height and so on, can be referred to by special names. At instantiation of a box, additional constraints can be added to those already in the definition, which together uniquely specify the size, shape and location of that instance of the box.

Van Wyk [11] discusses the types of constraints that should be made available and lists the worst case complexity results for finding the solutions to various kinds of constraints (linear equations, linear inequalities, linear equations with the operator max, quadratic equations, cubic equations in integers with linear inequalities, polynomials with the sine and absolute value functions). He arrives at the conclusion that moving beyond linearity quickly becomes computationally inefficient, with some steps simply being impossible in general as they would require solving undecidable problems.

Van Wyk [11] further notes that higher-order polynomials, in contrast to linear ones, result in more than one possible solution, raising the question of which of the solutions is to be selected. He concludes that simplicity, by only including the most useful forms of equation systems, seems wise.

Constraints in IDEAL are allowed to appear as something Van Wyk [11] calls *slightly nonlinear* systems of equations. He describes slightly nonlinear systems of equations as those systems of equations for which "there is an order in which the equations can be processed such that, after substituting results known from previous processing, the equation looks linear." These types of systems will be examined closer in later sections.

IDEAL's equation solver is similar to METAFONT's, but IDEAL's is more general. METAFONT requires the equations to be presented in order so that each equation can be made linear as it is processed, while IDEAL maintains a queue of equations to be retried after processing remaining equations, which removes the ordering requirement.

2.3 Juno

Juno [9] is a graphics system developed by Nelson to unify the author's perceived contrasting virtues of two earlier programs, Draw and METAFONT. While Draw was suitable for drawing free-form and unconstrained shapes, METAFONT was well-suited for producing images with a constrained and hierarchical structure. The author wished to produce images with a combination of both approaches.

Juno provides both a language for describing pictures and interactive picture editing capabilities. Working with Juno, the user is given both a view for text input using the graphics description language, and a view of the corresponding

picture. The user can interactively edit the picture, which results in changes in the program text.

Nelson [9] identifies three key features of METAFONT: points are specified by declarative constraints, images are rendered by imperative painting commands parameterized by the previously mentioned points, and procedural abstraction is provided. The use of declarative constraints in Juno is derived from METAFONT, but while METAFONT only allows linear constraints, Nelson argues that linear constraints are too limiting as they preclude the important geometric predicates of parallelism and congruence, which correspond to quadratic equations. Parallelism and congruence on pairs of points are in fact the only two constraint types that are provided in Juno. It is noted though that this is sufficient as any constraint expressible in Euclidean theory of geometry is expressible in terms of congruence and parallelism. Additionally, constraints for horizontal and vertical alignment of points are provided for initial orientation.

As parallelism and congruence translate to quadratic equations, numerical solving methods are employed. The method used is Newton-Raphson, which requires the solver to be provided with starting values in the vicinity of the target. These values must be provided by the user, for example using a pointing device in the interactive editor.

Nelson [9] identifies two issues with the use of nonlinear constraints: performance of the constraint solver, related to the numerical solving methods employed, and user-perceived unpredictability of results, related to the multiple solutions nonlinear equations result in. It is reported though that acceptable performance is possible to achieve, for example using standard numerical methods optimizations. Furthermore the problem of multiple solutions can be avoided by requiring users to place out points close to the expected solutions. These points can then be used by the solver as starting values to converge on the correct solution.

2.4 Cassowary

Badros and Borning [1] note in a paper presenting Cassowary the use for linear equality and inequality constraints in specifying user interfaces. Cassowary is an incremental constraint satisfaction algorithm suitable for use in specifying user interfaces.

A focus of Cassowary is that of efficiency, required for responsiveness in interactive applications. This is achieved through incremental updates exploiting previous computations. It is noted that efficient methods for solving constraints where the system of constraints is acyclic already exist, but as cycles often occur in real-world problems, they should be allowed. Therefore linear programming techniques, based on the simplex algorithm, which allow cycles are employed. One of the authors had experimented with using a pure simplex package, which had turned out to be a too inefficient solution. Therefore an efficient adaptation of the simplex algorithm was needed.

The objective in linear programming is to find values for a collection of real-valued variables, constrained by a collection of linear equality or inequality constraints, that minimize a linear objective function. The authors identify incrementality and defining a suitable objective function as the two principal issues in adapting the simplex algorithm.

Badros [2] identifies three primary limitations of Cassowary: it can handle only linear constraints, cannot handle disjunctions of constraints, and its domain is limited to numeric variables. It is recognized that the limitation to linear constraints is somewhat restricting, as for example Euclidean distance is nonlinear, but that for highly regular and aligned layouts rectilinear distances may suffice. Disjunctions are not provided by Cassowary, but were found to have been usable. The limitation to real-valued variables was recognized as the least restricting limitation, though extending to arbitrary domains would have enabled more of a system to be described declaratively.

Cassowary has been employed in a number of applications. Scheme Constraints Window Manager is a window manager for the X windowing system, Constraint Cascading Style Sheets is a constraint-based extension of CSS and Constraint Structured Vector Graphics extends SVG with constraints. It is worth noting that even though Cassowary was used in the Scheme Constraints Window Manager that is an application written in Scheme, the Cassowary implementation itself was written in C++ and had to be accessed through a foreign function interface.

3 Constraint Techniques

The general formulation of constraint satisfaction problems allows for arbitrary variable domains. In the context of graphics and geometry, constraint satisfaction problems usually boil down to constraining numeric variables, as seen in the section on related work. Even in systems where constraints are expressed in geometric terms such as parallelism and congruence, the constraints usually ultimately translate into constraints on numeric values. This results in the focus being put on constraint techniques that handle numeric variables.

A number of constraint techniques, primarily focused on constraining numeric variables, will be presented. This is intended to give a representative overview of the techniques employed in existing constraint-enabled figure drawing systems. With the exception of local propagation, all of these techniques are based on constraining numeric variables by means of mathematical equations.

Of the techniques presented here, mathematical equations, or more specifically, slightly nonlinear systems of equations, is the technique that has been employed in the constraint system presented later on.

3.1 Local Propagation

Local propagation is based on the concept of propagating feasible values in a network of variables dependent on (constrained by) each other. The value of each variable is decided by a function that calculates the value it must take, based on the values of other variables. These dependencies can be viewed as forming a directed graph where a dependency edge goes from a variable node to another variable node if the former variable constrains the value of the latter.

The value of each variable can at a specific moment in time be either known or unknown. If all variables that a variable depends on are known, the value of the variable can be calculated by applying the associated function. This calculation might in turn trigger the calculation of the value of another variable, causing a “propagation” of feasible values through the network of variables.

This construction of expressing constraints in terms of functions means that variables of arbitrary domains, including non-numeric ones, such as for example strings, can be used, as long as the value can be calculated as a function taking the values of other variables as input. The power of a constraint system depends on among others on available means to express these functions. In the extreme case, arbitrary Turing machines could be allowed, in which case local propagation would encompass all other constraint solving techniques (except perhaps human intuition).

Mappings of variables to functions that calculate their values can take different forms. There can for example be one-way or multi-way constraints. A one-way constraint maintains the value of a single variable, while a multi-way constraint in general can calculate the value of any of the variables it constrains, in terms of the other variables. Multi-way constraints are more general and provide a clearer way to specify relationships, and are therefore preferable.

Advantages of local propagation are that it is efficient, general, conceptually simple and that it can be used with arbitrary domains, as long as the value of a variable can be described as a function of other variables.

A major disadvantage of local propagation is that it cannot handle cycles in a constraint network, i.e., constraints that must be solved simultaneously. An example of a cycle is when the calculation of variable a requires the value of b, but for the value of b to be calculated the value of a is needed, and no progress can be made. This limitation is a result of local propagation only looking at one variable at a time, which is what the "local" in local propagation stands for. Local propagation solvers can be, and have been, extended to deal with cycles, usually by escaping to specialized cycle solvers.

3.2 Mathematical Equations

Mathematical equations provide a way to express constraints on the values a numeric variable can take. This is done by specifying an equation that includes the variable to be constrained and requiring the equation to be satisfied. That is, all values of the variable for which the equation is satisfied are allowed values. The concept can be generalized to a set of variables in a set of equations, where any set of values for the variables which simultaneously satisfies the set of equations is a solution.

Provided equations are chosen for expressing constraints, a closer look at the concept and properties of equations, from the point of view of methods to represent and solve these equations, is needed. Equations relate two expressions, the left-hand and right-hand sides, to each other. Equations are satisfied if the two expressions evaluate to the same value for a given assignment of values to the variables that are part of the the expressions. Expressions in turn are built up of combinations of mathematical operators, with constants and variables as operands. The process of solving systems of equations aims at finding assignments of values to variables for which all equations in a set are satisfied simultaneously. This is the task of the constraint solver in a constraint system that expresses constraints by means of mathematical equations.

Systems of equations can be divided into classes, based on the mathematical operators and combinations thereof that the equations are built up of. The division into classes is done as they are related to the solving methods available. As a constraint system must be able to solve the systems of equations presented

to it, there must be a suitable match between classes of equations used for expressing constraints, and solving methods employed by the constraint system.

An important property separating systems of equations built up of the elementary algebraic operators multiplication and addition into different classes based on available solving methods is the linearity of equations. Equations consisting entirely of terms that are products of a constant coefficient and a variable, with a potential additional constant-only term, are called *linear equations*. In contrast, equations that are not linear equations are *nonlinear equations*.

Below, a number of different types of systems of equations, including linear and nonlinear, and solving techniques related to them are discussed.

3.2.1 Linear Equations

Systems of linear equations contain only polynomials that are linear in the variables. They are advantageous in that they can be solved analytically, and that the algorithms for doing so are well-known and relatively simple. Gauss-Jordan elimination is one such algorithm.

A disadvantage of employing linear equations for constraints in graphics contexts is that they preclude quadratic equations that frequently arise in geometry. Most notably, expressing the distance between two arbitrary points in a plane requires quadratic equations. Nevertheless, linear equations are useful in order to, for example, constrain values in schematic figures built up around rectilinear patterns.

3.2.2 Nonlinear Equations

Nonlinear equations arise for example in Euclidean geometry, where the distance between two points in a plane give rise to an equation of second degree. Since many other relations in geometry are based on distance, this means that nonlinear equations are a desirable feature.

No general method for solving systems of nonlinear equations is known, and is not believed to exist [10]. Numerical methods exist that approximate solutions to systems of nonlinear equations, but they come with a number of disadvantages compared to analytical methods. The most obvious disadvantage is that numerical methods only produce approximations. Other disadvantages can be the requirement of a starting point, and lower performance.

The required starting point can in certain circumstances be taken advantage of though, as it enables disambiguation between multiple solutions. An additional advantage of certain iterative numerical methods, such as relaxation, is that they can be used to find "solutions" to over-constrained problems [8]. This is so because relaxation simply tries to minimize a continuous error function, based on to what degree constraints are satisfied, and doesn't require all constraints to be fully satisfiable.

3.2.3 Linear Equations with an Objective Function

Linear programming is the process of solving a system of linear equalities and inequalities over a set of unknown real variables, along with a linear objective function to be optimized. Constraining variables using linear programming methods is similar to constraining using systems of linear equations, with the

addition of allowing inequalities and an objective function to minimize or maximize. Linear programming, compared with the special case of solving systems of linear equations, provides more functionality at the cost of more complex solving algorithms.

Linear programming is a heavily studied problem with well-known solving techniques available, with the simplex algorithm being the most commonly used. The simplex algorithm first finds a feasible solution to the system, after which a process of systematically improving the solution begins. This process proceeds by in each step improving the solution, until the solution no longer can be improved. At this point a global optimum for the objective function has been found and the task has been completed.

3.2.4 Slightly Nonlinear Equations

The term "slightly nonlinear" is used by Van Wyk [11] to describe systems of equations for which there exist an order in which the equations can be processed that enables the system to be solved using standard linear equation system solving methods, while still allowing nonlinear elements to be included.

The nonlinear elements in such equations can be thought of as functions that take a number of variables as input, and produce a numeric value as output. No particular knowledge of the functions, or their inverses, are required, except for algorithms to compute them. These kinds of systems are solvable by in each step utilizing results from previous steps to simplify remaining nonlinear equations to linear form, eventually yielding linear equations solvable by standard methods.

Following is an attempt at a formal definition of systems of slightly nonlinear equations. A system of equations EQ is slightly nonlinear if one of the following two holds:

- EQ consists entirely of linear equations.
- Both of the following hold:
  - EQ has a subsystem $EQ' \neq \emptyset$ of linear equations that has a unique solution.
  - Let σ be the substitution that solves EQ'. Then $\sigma(EQ - EQ')$ is slightly nonlinear.

The solution to a slightly nonlinear system of equations, if one exists, is the union of the substitutions σ found during the process of simplifying remaining equations to linear form, plus the substitution that solves the final remaining set of linear equations.

If limited to numeric variables, it seems that any constraint network solvable using local propagation can also be expressed (and, thus, solved) using slightly nonlinear systems of equations. In local propagation, each node of the network is connected to the nodes that it depends on for computing its value. Associated with each node is a function that from the dependencies calculates the value of the variable represented by the node. This can be seen as an equation, with the variable that the node represents as the left hand side, and the function calculating the value of the node with the nodes dependencies as input as the right-hand side. The order in which node values are computed by local propagation yields a possible order in which respective equation can be processed according to the

requirements of a slightly nonlinear system of equations. Thus, the set of those equations is slightly nonlinear. If this observation is correct, it might be so that the technique presented by Van Wyk to solve slightly nonlinear systems of equations can solve the same class of problems that a local propagation solver could solve, if limited to numeric variables. In addition, Van Wyk's technique, being based on solving systems of linear equations, inherently handles cycles involving linear equations.

4 A Constraint System

Following the objective of this work, a constraint system for use with Functional PostScript has been created. The design and use of the constraint system will be presented here with the intent to show how constraint techniques can be leveraged in graphics systems. This includes presenting objectives behind the design, the underlying model of the constraint system core, important data types and abstractions constructed on top of the core constraint solver.

4.1 Objectives

A virtue of the project has been to keep things simple, and not investing time in implementing unnecessarily complex features. The employed constraint technique should be reasonably easy to implement, so that focus can be put on what it facilitates, rather than on the implementation itself.

The implementation of the constraint system itself together with the interfaces and anticipated usage patterns should as far as possible work in the spirit of the components they are to be used with. What is meant with this is that they should as far as possible honor the functional style of the main graphics system target, namely Functional PostScript, and the implementation language Scheme.

Even though Functional PostScript is the main target system in the scope of this project, it is desirable, if possible, to deliver a constraint system general enough to be used in other contexts. This includes isolating the system in independent modules, without graphics system specific dependencies.

It is anticipated that in addition to the core solver of the constraint system, a number of abstractions will be constructed on top of it to make the system more practical to use. The library nature of the system suggests that programmatic access to the primitives of the system should be provided to allow alternative abstractions to be constructed, in addition to the provided defaults.

The combination of the focus being on exploring the functionality of constraint systems and the target being a non-interactive environment, means that performance issues are of less priority. Targeting an interactive environment requiring high responsiveness would require more focus to be put on this question though.

4.2 Choice of Constraint Techniques

To explain the rationale behind the choice of the constraint technique, a discussion of the subject will follow here.

Working with a graphics system where positioning and other properties of objects are expressed in terms of numbers, constraining at the level of the numeric values is an obvious first approach, as is shown by the domination of mathematical constraint techniques in similar systems. To be as general as possible in what constraints can express, nonlinear equations should be allowed. However, in addition to the disadvantages of numerical methods mentioned earlier, it perhaps does not even make sense employing numerical methods in an environment where no practical means of acquiring starting values, required for numerical methods, exist. Therefore other alternatives have to be considered.

The generality and effectiveness of local propagation makes it an attractive candidate. It has the major disadvantage though, that it in its plain form does not handle cycles in dependencies, which is a severe limitation. It would be possible to complement local propagation with a cycle solver, which in the case of numeric variables would be a solver of systems of mathematical equations. As systems of mathematical equations can by themselves be employed as a constraint technique, implementing a cycle solver for local propagation would actually mean implementing two constraint techniques, going against the objective of keeping the implementation simple.

An alternative is to use mathematical equations only, of a class for which general and relatively easily implementable solvers exist. Linear equations is a class for which such solving methods exist, making it an attractive choice. However, it is relatively easy to increase the usefulness of a system using linear equations by extending it to handle slightly nonlinear equations.[2]

This makes slightly nonlinear equations an even better choice. The fact that two other systems targeting similar use (METAPOST and IDEAL) have employed slightly nonlinear equations, or variations thereof, themselves, suggests that others might have arrived at similar conclusions regarding the appropriateness of slightly nonlinear equations for the task.

As noted in the previous section, it seems that much, if not all, of what can be expressed and solved using local propagation techniques, can be solved using slightly nonlinear equation techniques. Regardless of this, slightly nonlinear equation techniques stand on their own merits. Therefore they have been chosen as the constraint technique to base the constraint system on.

4.3 Expressions and Equations

Given that constraints are expressed using equations, the data type for representing mathematical expressions, of which equations are constructed, is central to the constraint system. Expressions have a tree structure, in which each subtree also is an expression. The internal nodes of the tree are operators, and the leaf nodes variables and constants. The constraint system provides procedures for constructing such trees to represent arbitrary expressions.

There are three low-level expression construction procedures: a *composite expression* constructor, a *variable expression* constructor and a *constant expression* constructor.

`(make-oexpr` *oper args*`)` procedure

[2]Formally, it is not correct to speak of "slightly nonlinear equations," as the property of being "slightly nonlinear" really is the property of a *system* of equations. It is however a convenient term for equations that are part of a "slightly nonlinear" system of equations, which is how it will be used hereafter.

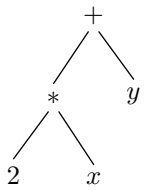

Figure 3: Tree interpretation of expression $2x + y$

`(make-vexpr` *ident*`)` procedure
`(make-cexpr` *value*`)` procedure

The `make-oexpr` procedure is the composite expression constructor. Composite expressions consist of an operator and a number of arguments. The operator argument is a Scheme procedure that takes a number of numeric arguments and calculates the value resulting from applying the operator it represents to those arguments. The arguments argument is a list of expressions that at expression evaluation time will be evaluated into numeric values and passed to the operator procedure.

The `make-vexpr` procedure is the variable expression constructor and creates an expression representing the single variable that the identifier argument identifies. The identifier argument is a value that uniquely identifies the variable among a set of variables used in the same context. The value can be an arbitrary Scheme value, as long as two different identifiers do not yield true for the Scheme `equal?` predicate.

The `make-cexpr` procedure is the constant expression constructor and creates an expression representing a single constant numeric value given as its argument.

```
(make-oexpr +
 (list (make-oexpr * (list (make-cexpr 2) (make-vexpr 'x)))
       (make-vexpr 'y)))
```

This is an example of how the expression $2x + y$ could be constructed using the low-level expression constructors. The corresponding tree interpretation of the expression is illustrated by Figure 3.

Equations consist of expressions, namely the left-hand and right-hand sides. As equations consist of expressions, expressions are used as input to the equation constructor.

`(make-eqn` *lhs rhs*`)` procedure

The `make-eqn` procedure is the equation constructor and takes as its arguments the expressions denoting the left-hand and right-hand sides of the equation.

```
(make-eqn
 (make-vexpr 'x)
 (make-oexpr * (list (make-cexpr 3) (make-vexpr 'y))))
```

This is an example of how the equation $x = 3y$ could be constructed using the low-level equation constructor.

2	`2`
x	`x`
$2x$	`(* 2 x)`
$2x + y$	`(+ (* 2 x) y)`

Table 1: Examples of embedded expression notation

$\sqrt{x}$	`(sqrt x)`
x^2	`(expt x 2)`
$\max(x, y)$	`(max x y)`

Table 2: More examples of embedded expression notation

Using the low-level procedures presented here is not a practical way for end users of the system to create expressions or equations, due to the code verboseness. Instead, more practical and compact notations that translate to calls to these low-level procedures will be in the following section. These low-level procedures are however exposed for completeness regarding access to the core data type of the system. This allows users to create alternatives to the abstractions bundled with the constraint system.

4.4 Embedded Expression and Equation Notation

As using the low-level expression constructors is not practical, alternatives have to be provided. A convenient and compact way of representing expressions is facilitated by Scheme's capability of embedding structured data in code. The constraint system provides a syntax keyword that transforms a notation, reminiscent of the prefix notation used in evaluating mathematical expressions in Scheme, into instances of the expression data type. This notation is called the *embedded expression notation*.

`(expr` *expr*`)` syntax

The `expr` syntax keyword transforms its argument into calls to the low-level expression constructors to construct the denoted expression. The argument can either be a list representing a composite expression, a symbol representing a variable, or a number representing a constant. In a list that represents a composite expression, the first element denotes the operator, while the rest of the elements denote the associated arguments. The arguments are themselves in turn expressions in the same notation, making the notation recursive. For a variable, the symbol is used as the identifier. Examples of expressions and their corresponding notations are shown in Table 1.

The symbol in the operator position of a composite expression is evaluated in the Scheme environment to produce the actual Scheme procedure used for the calculations of the operator. This means that the standard numeric operators provided by Scheme are available for use in expressions by default. Therefore it is possible to construct expressions such as those shown in Table 2. Note that even though the notation allows arbitrary number-to-number operators, systems of equations are subject to the requirements placed by the definition of

$x = 2$	`x 2`
$x = 2y$	`x (* 2 y)`
$2x = 3y + 4$	`(* 2 x) (+ (* 3 y) 4)`

Table 3: Examples of embedded equation notation

slightly nonlinear systems of equations to be solvable by the constraint solver.

Closely related to the embedded expression notation is the *embedded equation notation*. As equations consist of expressions the notation for equations builds on the notation for expressions.

`(eqn` *lhs rhs*`)` syntax
`(eqns (`*lhs rhs*`) ...)` syntax

The `eqn` syntax keyword transforms its two arguments denoting the left-hand and right-hand side expressions of an equation into calls to the low-level expression constructors to construct the denoted equation. The `eqns` syntax keyword is a convenience construct that transforms pairs of left-hand side and right-hand side expressions into calls to construct sets of equations. Examples of equations and their corresponding notations are shown in Table 3. Note that the mandatory use of parenthesis to denote composite expressions makes the use of equal signs redundant. Equal signs are not used in the notation at all.

4.5 Solving

The core of a constraint system is the *constraint solver*. It takes as input a set of constraints on variables, and produces as output the acceptable values for the constrained variables under the given constraints. Constraints are here expressed using equations, and handled in sets. Sets of equations are represented using Scheme lists, with individual equations constituting the elements of the list.

`(solve` *equations*`)` procedure
`(inspect` *ident solution*`)` procedure

The `solve` procedure returns the solution for a slightly nonlinear set of equations. The solution is an association list mapping variables to their numeric values. In case the system of equations is inconsistent (over-constrained), a unique solution does not exist (under-constrained), or not slightly nonlinear, an error message to that effect is generated and further processing halted. For closer details on the solving algorithm, see Section 5.2.

The `inspect` procedure extracts the value of the variable identified by the given identifier from a given solution.

```
(inspect 'x
 (solve (eqns ((+ (* 2 x) y) 4)
              ((+ (* 3 x) y) 5))))
```

This example shows how the value of a variable is inspected from the solution produced by applying the solver procedure on a set of equations.

4.6 Embedding Equations in Arbitrary Code

The facilities of the constraint system presented so far consist of the core solver and a practical notation for expressing constraints. Use of these alone in an application requires a focus on the constraint solving process, that likely interferes with the natural flow of the application. This is so because the process of gathering equations, applying the constraint solver to the equations and extracting values from the solution has to be performed explicitly. To allow for constraint techniques to be integrated more smoothly, facilities for making the constraint solving process more implicit are needed.

This is here done in the form of a set of syntax keywords, that enable any expression in arbitrary Scheme code required to evaluate to a number to be expressed in constraint terms. A body of code is regarded as a context in which a number of equations occur. Anywhere in this context values of variables occurring in these equations can be substituted for locations where numbers are required to occur in the code. What essentially is done, is that the code is put through a preprocess phase where the equations are gathered, the system solved, and the numeric values from the solution inserted into the code at the appropriate places.

`(with-embedded-equations` *body*`)` syntax

The `with-embedded-equations` syntax keyword collects equations found in its body, solves the resulting system, and substitutes values from the solution into the body at the appropriate locations.

`(var` *ident*`)` syntax
`(var` *ident expr*`)` syntax
`(var` *ident expr* `(`*lhs rhs*`) ...)` syntax

The syntax keyword `var` denotes a numeric value, expressed as a variable of the equation system consisting of all equations under the most closely nested `with-embedded-equations`. It takes the identifier of the variables whose value is to be substituted for the form. If the value of the variable is uniquely defined by the rest of the equations, no further information is needed. If however that is not the case, an expression may be included that will translate to an equation `(eqn ident expr)` using the embedded equation notation, and uniquely define the value. In addition, an arbitrary number of equations in the form `(lhs rhs)` may be included, that translate to `(eqn lhs rhs)` and are included in the system.

```
(with-embedded-equations
 (display (var x (* 2 y))) (newline)
 (display (var y (* 2 z))) (newline)
 (display (var z (- x 3))) (newline))
```

This example executes three sequential display operations, to display the values of `x`, `y` and `z`, which are 4, 2 and 1 respectively. No explicit calls to the solver are required. Note how the values to be printed are expressed in relation to values occurring only later in the code.

This facility is implemented using macros, to show what is possible. Macros should be used with care though as they complicate understanding of code. It is left to the reader to judge the value of this construct.

4.7 Namespaces

The usefulness of the constraint system can be increased by recognizing usage patterns and building abstractions above the core system to support these patterns. An abstraction that will prove to be important in abstractions presented later on is that of *namespaces*, which applies to the naming of variables.

Namespaces provide naming of variables in multiple levels. The constraint solver itself is oblivious to the representation of variable identity, as long as two distinct variable identifier values do not yield true for the Scheme `equal?` predicate. Therefore, it is possible to construct arbitrary naming schemes, for example by using structured data to represent variable identity.

In the multi-level naming provided by the namespaces abstraction, each level identifies a subset of variables sharing a common namespace. Each level narrows down the subset of variables, with the last level identifying distinct variables, organizing the variables into a hierarchy. This allows sets of related variables to be embedded into larger sets while preserving the identity of the inserted set.

The namespace abstraction maintains a *current namespace stack*, a dynamically scoped value representing the prefix that fully qualifies a variable identifier.

`(with-namespace` *namespace* `.` *body*`)` syntax
`(with-absolute-namespace` *namespace* `.` *body*`)` syntax

The `with-namespace` and `with-absolute-namespace` syntax keywords manipulate the current namespace stack for the duration of the evaluation of the body. The `with-namespace` pushes the given namespace onto the stack for the duration of the evaluation of the body, while `with-absolute-namespace` replaces the stack with the one-element stack consisting of the given namespace for the duration of the evaluation of the body.

`(qualify` *ident*`)` procedure
`(qualify-namespaced-symbol` *symbol*`)` procedure

The `qualify` and `qualify-namespaced-symbol` procedures *qualify* a variable identifier with the namespace constructed from the current namespace stack. While `qualify` is the basic version that qualifies an arbitrary value, `qualify-namespaced-symbol` is a convenience procedure that takes a symbol as input, of which after a conversion to a string each successive dot separated prefix is converted back to a symbol and pushed onto the current namespace stack for the duration of qualifying the identifier.

The symbols denoting variable identifiers in the embedded expression and equation notation are implicitly treated to `qualify-namespaced-symbol` as a convenience.

```
(qualify-namespaced-symbol 'n.x)
(with-namespace 'n (qualify 'x))
```

This example illustrates the use of `qualify-namespaced-symbol` in terms of `with-namespace` and `qualify`. Both forms are equivalent in regard to the variable identifier they produce.

4.8 Dimensions

In figure drawing where positions are described using points—ordered tuples of values—the majority of relations are stated in terms of points. Therefore it is

desirable to be able to state those relations in single units, instead of breaking them up into separate relations, one for each dimension.

`(dimen` *suffixes expr*`)` procedure
`(dimen/2d` *expr*`)` procedure

The `dimen` and `dimen/2d` procedures derive from a single input expression (equation) a list of expressions (equations) according to a number of rules.

The `dimen` procedure is the general version that, in addition to an expression, takes as input a list of suffixes of which each is appended to variable identifiers in the corresponding output copy of the expression. For two dimensions the suffix list could for example be `'(x y)`, and for three dimensions `'(x y z)`. For the special case of an empty list, the input expression is returned unmodified. In addition, each occurrence of a vector (introduced by the `#(...)` syntax) is replaced by the contents of the vector at the index of the suffix for the corresponding output copy, allowing completely different expressions to be substituted into specific copies.

The `dimen/2d` procedure is a convenience procedure that is equivalent to the common case of `(dimen '(x y) expr)`.

`(eqn/2d` *lhs rhs*`)` syntax
`(eqns/2d (`*lhs rhs*`) ...)` syntax

In addition to the plain `eqn` and `eqns` syntax keywords used with the embedded equation notation, corresponding convenience versions `eqn/2d` and `eqns/2d` that apply `dimen/2d` to the equations are provided.

```
(dimen/2d
 (eqn p1 (+ p2 #(10 20))))
```

This example shows how `dimen/2d` can be used to expand a single equation, where each variable denotes a two-dimensional point, into two equations, where each variable denotes a number.

```
(list
 (eqn p1.x (+ p2.x 10))
 (eqn p1.y (+ p2.y 20)))
```

This is the corresponding version that explicitly creates two equations, using the implicit interpretation of embedded namespaces in symbols done by the embedded equation notation.

4.9 Templates

The *templates* abstraction allows constraint data to be packaged for reuse. Constraint data in the case of a constraint system such as the one presented here, using mathematical equations to constrain variables, are equations. Templates in such systems therefore consist of sets of equations.

The templates abstraction actually goes slightly beyond managing constraint data. The constraint data in templates as presented so far is usually closely coupled with a procedure that takes as input a solution that satisfies the constraints in the template and produces a graphical interpretation of that. Therefore templates come with, in addition to a set of constraint data, such a procedure that graphically interprets a solution that satisfies the constraints of the template.

Namespaces play an important role in the templates abstraction as they facilitate merging equations of different template instances into single sets as is required for the solver, while still allowing variables of specific template instances to be identified in the merged set.

`(template` *eqns prod*`)` procedure
`(compose` *comb tpls eqns*`)` procedure

The `template` and `compose` procedures allow for the creation of templates. The `template` procedure is the general version that takes as arguments a procedure that produces the equations of the template and another procedure that produces a graphics interpretation of a solution that satisfies the set of constraints of the template.

The `compose` procedure creates a template by composing other templates. The first argument is a procedure that combines the graphics interpretations of the *child templates* into a single graphics object. The second argument is an association list mapping identifiers to templates, representing the child templates that the template is to be composed of. The third argument is a list of equations that augment the equations of the child templates, creating a system with a unique solution.

A number of template primitives with graphical interpretations in Functional PostScript are provided for use in composing templates. These are: `line` for lines, `rect` for rectangles, `curve` for Bezier curves, and `arc` for arcs. The lines template takes a number of points, two at a minimum, named `p1`, `p2`, `p3` and so on, and draws a line through them using straight line segments. The rectangles template takes a combination of the corner points and side midpoints named after the compass points (south-west, south-east, north-east, north-west, south, east, north and west), the center point, and the delta (the difference between the south-west and north-east points) that uniquely defines a rectangle, and draws that. The curve template takes a starting and ending point, and two control points, and draws a Bezier curve using them. The arc template takes a center point, a radius, and a starting and ending angle, in radians, and draws an arc using them. The names of the properties used by the templates are shown in Figure 4.

`(instantiate` *tpl eqns*`)` procedure

The `instantiate` procedure instantiates a template, returning the graphical interpretation. This is done by passing the equations of the template together with augmenting equations to the solver that produces a solution, which then is passed to the template's graphical interpretation procedure to produce the graphical interpretation.

The flow of data between the components of the constraint and graphics system when using templates is illustrated by Figure 5. Templates provide sets of equations that together with augmenting equations are passed to the solver to find a solution, which in turn is passed to the template's associated procedure for producing a graphical interpretation of the solution, which in turn is passed to the underlying graphics system to render the graphical interpretation.

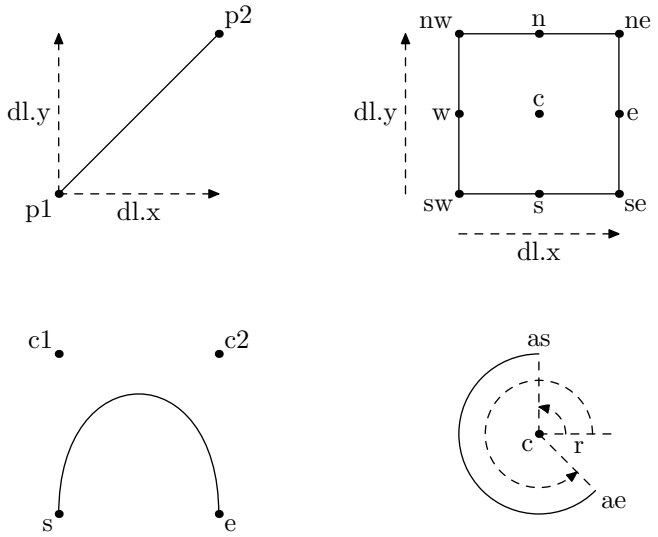

Figure 4: Template interfaces

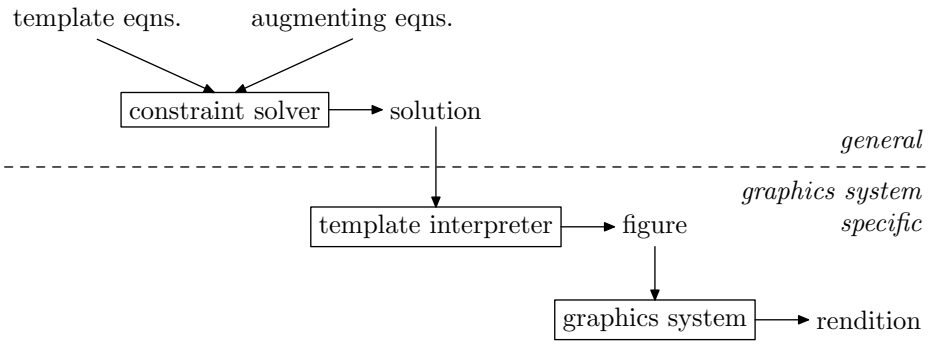

Figure 5: Data flow when using templates

4.10 Integration

The question of how and how far the constraint system and the graphics system are to be integrated is important. Having numeric values as the domain of variables and using equations to express constraints has the fortunate property of being very general, with nothing graphics specific at all. This enables the core constraint system to be treated as a fully independent module, with abstractions involving graphics system specific parts to be stored in separate and isolated modules.

4.11 Limitations

The constraint system as described here requires the existence of a unique solution, which requires careful crafting of the system of constraints that constrain the solution. It is a problem that users easily inadvertently over- or under-constrain the solution. Redundant constraints do not affect the solution, and can therefore silently be discarded during processing. In contrast, over-constrained systems have conflicting constraints, leaving an empty solution space, and under-constrained systems multiple, possibly infinitely many, solutions, leaving the system unable to arbitrarily select one. The constraint system in its current state is not able to provide much feedback on the cause of not being able to arrive at a unique solution. This is indeed frustrating to the user, who has no option other than to manually reinspect the entire input to the solver.

In all cases, the exceptional condition of an over- or under-constrained system is reported. The question is if the constraint system could derive a useful unique solution from over- or under-constrained systems, by somehow guessing the intention, or provide error messages pointing out the cause of the problem more closely. The constraint system presented here does not attempt to further process over- or under-constrained systems. Constraints are required to narrow down the solution space to a unique solution to give the system reliable and predictable behaviour, and to simplify implementation.

5 Implementation

While the design and use of the constraint system has been presented in the previous section, notes on the implementation will be presented here.

5.1 Data Types

The data type for representing mathematical expressions, occurring as the left and right-hand sides of equations, is a central data type of the system. The most general way to represent expressions would be something that corresponds to the description given in Section 4.3, i.e., as a tree with internal nodes consisting of operators and leaf nodes consisting of variables and constants. However, the solving algorithm operates on a level slightly above individual nodes of the tree. Due to the centrality of the solving algorithm in the system, the internal representation of an expression is modeled for the convenience and efficiency of the solving algorithm.

Expressions are considered to be built up of a set of terms, categorized into three different types, with each type being stored in a separate list. The three types are function terms, variable terms and constant terms. Function terms consist of a function represented by a Scheme procedure and associated arguments. Variable terms consist of a coefficient and a variable identifier. Constant terms consist of a constant numeric value. The difference to the general tree structure lies here in that the entire set of terms are implicitly joined by an addition operator, and that the variable terms' coefficients and variables are joined by a multiplication operator. This allows various types of subexpressions significant to the solving algorithm to be distinguished efficiently.

There is no separate data type for equations. Instead, equations are represented in the form of expressions. These expressions represent the left-hand sides of equations rewritten to have the right hand-sides equal zero.

The support for numbers in Scheme includes a so-called tower of numeric subtypes, where each level is a subset of the level above it: numbers, complex, reals, rationals and integers. A distinction is made between *exact* and *inexact* numbers; computations involving exact numbers produce mathematically equivalent results across implementations, while computations involving inexact numbers may not produce equivalent results across implementations, as approximate methods such as floating point arithmetic may be used. As a consequence of the constraint system relying on the Scheme number support for operations on numbers, these rules also apply to calculations performed by the constraint system.

5.2 Solver

The core of the constraint system—the equation solver—solves slightly nonlinear systems of equations, as described in Section 3.2.4. Solving systems of slightly nonlinear equations is closely related to solving systems of linear equations, a well-understood problem with known solving methods.

Elimination and substitution are two methods by which systems of linear equations can be solved. Both systematically strive to remove variables in equations to ultimately end up with an equation in a single variable. The single remaining variable is then trivial to solve for, and can be used in a sequence of back-substitutions to reveal the values of the rest of the variables. In elimination, variables are eliminated by adding equations to each other in suitable combinations. In substitution variables are eliminated by substituting equivalent expressions in terms of other variables for the variables to be eliminated.

Substitution suits well for use in solving slightly nonlinear systems of equations. The constraint system uses an algorithm based on substitution inspired by the algorithm described and used by Van Wyk [11] in IDEAL.

The algorithm works by performing several passes over the set of equations in the system, as described in Algorithm 1. Each pass uses partial results, in the form of values of variables which have been solved for so far, from the previous passes to evaluate and simplify nonlinear equations into linear equations. New passes are made over the equations as long as iterations reveal values of new variables. Each pass involves processing each equation once. During each pass a set of dependent variables and their expressions in terms of independent variables are built up and used in the processing of subsequent equations in the same pass.

Algorithm 1 Solving Systems of Slightly Nonlinear Equations

```
repeat
  for all equations do
    substitute dependents
    normalize (evaluate and simplify) equation
    if linear equation then
      if variables in equation then
        choose new dependent
      else {constant equation, meaning inconsistency or redundancy}
        if inconsistent then
          terminate with error
        else
          ignore redundant equation
        end if
      end if
    else
      ignore nonlinear equation
    end if
  end for
until an iteration didn't reveal the value of a new variable
```

If the system of equations is slightly nonlinear as defined in Section 3.2.4, the solver will find the unique solution, if one exists. During each pass, the solver uses the substitution technique to solve as much as possible of the subset of linear equations in the system. Assuming the system is slightly nonlinear, these results can then be used to simplify remaining nonlinear equations into a new slightly nonlinear system, after which the process is repeated, eventually leading to a solution of the entire system. If the solver comes to a point where the remaining set of equations are all nonlinear, the system is not slightly nonlinear, which is reported as an error.

The algorithm of the solver is possible to implement with a relatively small effort.

5.3 Limitations

The current implementation has acceptable performance for solving smaller problems. However, for larger problems performance enhancements would be required for the system to be of practical use.

6 Examples

The constraint system, including the abstractions built upon it and presented in previous sections, can be used in a number of ways. Here, a few simple examples will be presented.

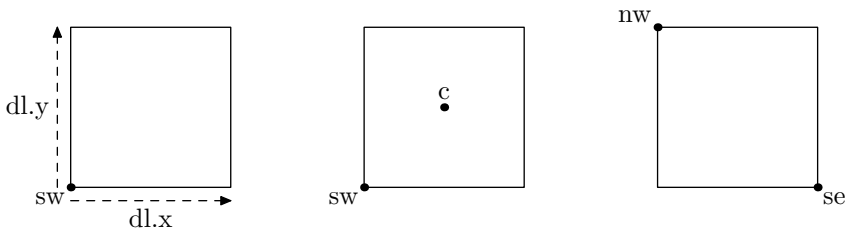

Figure 6: Same rectangle defined in multiple ways

6.1 Defining a Rectangle in Multiple Ways

Among the templates provided by the system is one for rectangles. It defines equations that relate various properties of rectangles to each other, such as the locations of the center and corner points, and the height and width. This makes it possible to uniquely define the same rectangle in a number of ways, by varying the subsets of properties used. Properties that have not been assigned values explicitly have their values inferred from the properties that have been assigned values explicitly, using the equations of the template.

Below are three different instantiations of the rectangle template that produce the same rectangle (lower-left hand corner at the origin, width 2 and height 2), but using different subsets of properties. The selection of properties used in each definition is illustrated by Figure 6.

```
(instantiate rect (eqns/2d (sw #(0 0)) (dl #(2 2))))
(instantiate rect (eqns/2d (sw #(0 0)) (c  #(1 1))))
(instantiate rect (eqns/2d (nw #(0 2)) (se #(2 0))))
```

All three make use of the `dimen/2d` procedure to expand the equations in points into equations in numbers, adding the `x` and `y` suffixes to variables and splitting contents of vectors (introduced by the `#(...)` syntax) into appropriate dimensions.

The first uses the normal form of specifying the location of the south-west corner together with the height and width, the second specifies the south-west corner and the center, while the third specifies the north-west and south-east corner. All three produce the same rectangle graphics value.

6.2 External Template Notation

The embedded equation notation provides a way to embed constraint data in Scheme code. There are however situations when one might want to store larger collections of constraint data externally, for modularization and reuse. Templates provide an abstraction suitable for handling sets of constraint data, but an external notation is needed. As the data is stored in an external file, as opposed to being embedded in Scheme code, it can be of arbitrary form. In addition to notation for template concepts, a notation for equations is needed. The embedded equation notation has several advantages, especially for Scheme-savvy users. Everyone is however not comfortable with the s-exp based prefix notation. It would be of interest to enable users that are not acquainted with

Scheme to compose templates in an external notation. For this purpose the more common infix notation is suitable.

A system for parsing an external notation of templates has been created on top of the core solver and the namespaces, dimensions and templates abstractions. The system is built up around parsing plain text files and translating the contents into appropriate calls to the constraint system abstractions according to the descriptions below. It shares several similarities with IDEAL's [11] notation and concept of "boxes."

`(parse-files` *prims comb files*`)` procedure

This procedure parses files with templates in the external template notation, returning an association list mapping names to templates. The procedure takes as input an association list of initial template primitives, a procedure for combining components of a template and a list of file names to parse in the given order. While parsing, an association list mapping names to templates learned so far during the parsing process is maintained, which is initialized to the given initial template primitives.

Each file contains a number of blocks, representing templates to be created. Each block is introduced by a string identifying the template name, and contains enclosed in braces a number of child template name, identifier and augmenting equations triplets, each terminated by a semicolon, that represent the child templates that the template being defined is to be composed of. The child template name is used to lookup the template among the templates learned so far during the parsing process. The identifier is the namespace in which the variables of the child template are placed. The augmented equations, enclosed in braces, are added to the total set of equations. Additionally, the last entry of a template block can be a set of equations enclosed in braces without an associated child template and identifier.

Equations are specified in an infix notation, and terminated by a semicolon. Equations are by default expanded for two dimensions, as per the dimensions abstraction, using `x` and `y` for suffixes. This expansion can be inhibited by prepending an equation with an apostrophe (`'`). Separate expressions for each dimension can be specified with a comma-separated list of expressions enclosed in brackets (`[]`). Comments start with a hash (`#`) and end at the following newline character.

Following are three simple example uses of this notation. All three make use of composing from previously defined templates. The templates depended on are defined first, after which a "main" template is defined and used for generating the figures accompanying the definitions below.

6.2.1 Linked List

The figure idea here is to illustrate a segment of a "linked list." A linked list consists of nodes connected to each other through pointers. Each node contains, in addition to a pointer to the next node, data for one list element. A node could for example be illustrated by a rectangle block split into two parts, representing the data and pointer cells. To show where a pointer points, an arrow can be positioned pointing from one node to the next.

An arrow is a figure concept that suits well for being implemented as a reusable template. An arrow template composed of three lines is created. It

is controlled through the `t` (tail) and `h` (head) variables that specify where it starts and ends.

```
# interface: t (point), h (point)
arrow {
  # shaft
  line l {
    l.p1 = t;                   # tail (input)
    l.p2 = h;                   # head (input)
    'hl = 7;                    # head length
    d = t - h;
    'a = atan(d.y, d.x);        # shaft angle
  };
  # head
  line h1 {
    'a1 = a + 3.142 / 8;        # angle
    h1.p1 = h;                  # connect to head
    h1.p2 = [h.x + cos(a1) * hl, h.y + sin(a1) * hl];
  };
  line h2 {
    'a2 = a - 3.142 / 8;        # angle
    h2.p1 = h;                  # connect to head
    h2.p2 = [h.x + cos(a2) * hl, h.y + sin(a2) * hl];
  };
}
```

A linked list node is also a concept that can be implemented as a reusable template. It is composed of two rectangles and controlled through the `data` and `next` variables that are rectangles.

```
# interface: data (rect), next (rect)
node {
  rect data {
    data.dl = [30, 30];
  };
  rect next {
    next.sw = data.sw + [data.dl.x, 0];
    next.dl = data.dl;
  };
}
```

Finally, a "main" template is used to bring together instances of the arrow and node templates in a desired configuration.

```
main {
  # nodes
  node n1 { n1.data.sw = [0, 0]; };
  node n2 { n2.data.sw = n1.next.ne + [-5,  20]; };
  node n3 { n3.data.nw = n2.next.se + [-5, -20]; };

  # arrows
```

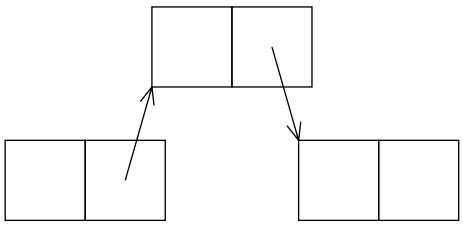

Figure 7: Linked list

```
  arrow a1 { a1.t = n1.next.c; a1.h = n2.data.sw; };
  arrow a2 { a2.t = n2.next.c; a2.h = n3.data.nw; };
}
```

The resulting output can be seen in Figure 7. The three nodes are successively positioned in relation to each other, as are the arrows to the nodes. This ensures that a change in the positioning of the first nodes is reflected in the positioning of the other nodes, and that the arrows move along together with the nodes.

The figure idea is borrowed from a paper on IDEAL by Van Wyk [11].

6.2.2 Four Boxes

The figure idea is to connect four boxes in a circular fashion. For this example no new templates need to be defined as the previously defined arrow template can be reused, and a rectangle template comes as a primitive template with the constraint system.

A "main" template is created, that instantiates four arrows, and the same number of rectangles.

```
main {
  # arrows
  arrow a1 { a1.t = r1.e; a1.h = r2.w; };
  arrow a2 { a2.t = r2.n; a2.h = r3.s; };
  arrow a3 { a3.t = r3.w; a3.h = r4.e; };
  arrow a4 { a4.t = r4.s; a4.h = r1.n; };

  # boxes
  rect r1 { r1.c = c + [-50, -50]; r1.dl = dl; };
  rect r2 { r2.c = c + [ 50, -50]; r2.dl = dl; };
  rect r3 { r3.c = c + [ 50,  50]; r3.dl = dl; };
  rect r4 { r4.c = c + [-50,  50]; r4.dl = dl; };

  # common
  { c = [70, 70]; dl = [40, 40]; };
}
```

The resulting output can be seen in Figure 8. As the boxes are to be placed out in a symmetrical fashion, all four can be placed out in reference to their

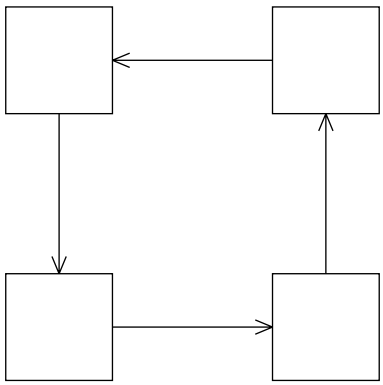

Figure 8: Four boxes

center points. When connecting the boxes with arrows, referencing other parts than the center points are needed. The rectangle template provides equations that relate the center points to the points needed, i.e., the south, west, north and east border points on a rectangle.

The figure idea is borrowed from a book on constraint programming languages by Leler [8].

6.2.3 Graph

The figure idea here is to show how a simple graph might be drawn. A graph consists of nodes and edges. For an undirected graph lines suffice to illustrate edges. Nodes could be represented by arcs. These are simple enough concepts, but are somewhat complicated if the edges are not to cross into the arcs representing the nodes. Edge and node templates are created.

```
# interface: p1 (point), p2 (point)
edge {
  line l {
    d = p2 - p1;                # input
    'nl = 10;                   # node radius
    'el = sqrt(expt(d.x, 2) + expt(d.y, 2)); # edge length
    'q1 = nl / el;
    'q2 = (el - nl) / el;
    l.p1 = [p1.x + q1 * d.x, p1.y + q1 * d.y];
    l.p2 = [p1.x + q2 * d.x, p1.y + q2 * d.y];
  };
}

# interface: c (point)
node {
  arc a {
    a.c = c;                    # input
    'a.r = 10;                  # radius
    'a.as = 0;                  # angle start
```

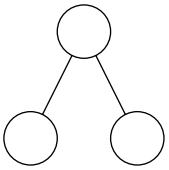

Figure 9: Graph

```
    'a.ae = 6.283;              # angle end
  };
}
```

The edge template makes use of square root and exponentiation operators to calculate the part of the line that should be visible to not cross into the node.

A "main" template is created, that instantiates three nodes and two edges to connect them.

```
main {
  # nodes
  node p1 { p1.c = [30, 50]; };
  node c1 { c1.c = p1.c + [-20, -40]; };
  node c2 { c2.c = p1.c + [ 20, -40]; };

  # edges
  edge e1 { e1.p1 = p1.c; e1.p2 = c1.c; };
  edge e2 { e2.p1 = p1.c; e2.p2 = c2.c; };
}
```

The resulting output can be seen in Figure 9.

7 Conclusions

It is possible to, in a high-level language such as Scheme, implement a core solver that solves slightly nonlinear systems of equations, with a relatively small effort. Through this relatively small effort a declarative approach to graphics, with the associated benefits, is made possible. To make a constraint system based on mathematical equations usable in practice, more than an equation solver needs to be made available. Suitable abstractions constructed on top of the core equation solver are needed.

Furthermore, it has become evident during the project that it is possible to employ constraint techniques in a way that allows implementing the constraint system as a general module, independent of a specific target such as the graphics system here.

References

[1] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

[2] Gregory Joseph Badros and Alan Borning. *Extending interactive graphical applications with constraints (user interface)*. PhD thesis, University of Washington, 2000.

[3] John D. Hobby. A user's manual for MetaPost. Computing Science Technical Report no. 162, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992.

[4] Adobe Systems Inc. *PostScript Language Reference (3rd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[6] Donald Ervin Knuth. *The METAFONTbook*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[7] Helge von Koch. Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes. *Acta Mathematica*, 30:145–174, 1906.

[8] William Leler. *Constraint Programming Languages: Their Specification and Generation*. Addison-Wesley, 1988.

[9] Greg Nelson. Juno, a constraint-based graphics system. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 235–243. ACM Press, 1985.

[10] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[11] Christopher J. Van Wyk. A graphics typesetting language. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 99–107, 1981.

[12] Wandy Sae-Tan and Olin Shivers. Functional PostScript home page, 1996. `ftp://ftp.scsh.net/pub/scsh/contrib/fps/doc/fps.html` (as of September 1, 2003).