# Developing a Stage Lighting System from Scratch

Michael Sperber
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
sperber@informatik.uni-tuebingen.de

## ABSTRACT

Lula is a system for computer-assisted stage lighting design and control. Whereas other systems for the same purpose are usually the results of long chains of incremental improvements of historic concepts, Lula represents a complete redesign. Whereas other systems focus on *control* aspects of lighting, Lula focuses on *design* and generates control information from it. This approach gives significantly more flexibility to the lighting designer and shortens the design process itself.

Lula's design and implementation draw from a number of disciplines in advanced programming. It is written in Scheme and runs atop *PLT Scheme*, and benefits from its high-level GUI library. Lula uses an algebraic model for lighting looks based on just three combinators. It employs Functional Reactive Programming for all dynamic aspects of lighting, and is programmable via a functional-reactive domain-specific language.

Lula is an actual product and has users who have neither interest in nor knowledge of functional programming.

## 1. THE DEMANDS OF STAGE LIGHTING

Live shows require lighting. This applies to theater, but also to concerts and industrial presentations. Lighting a stage well is surprisingly hard. Modern stage lighting prescribes multiple colored light sources for a single place on stage. Even though the complexity of the lighting is often not apparent to the spectator, the difference between white head-on lighting and a setup which takes into account modelling, focus, environmental representation, mood, temperature, etc. is striking. This makes lighting design a difficult craft and an art with recognized masters [17, 20].

The demands of the craft have produced a stunning arsenal of technology available to today's lighting designer: Modern intelligent lighting fixtures (or just *fixtures* for short) have electronic controls for intensity, direction, focus, color, beam shape, and many other parameters. Their practical use requires computerized control systems.


*ICFP'01,* September 3-5, 2001, Florence, Italy.


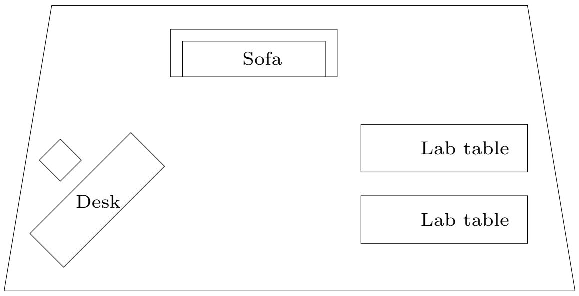


**Figure 1: Set groundplan for *Doctor Love*.**

*Lighting Looks.* Consider a simple lighting session in a theatrical setting. The basic ingredients are a stage with props and actors, and simple theatrical fixtures with variable intensity. The play is *Doctor Love*: Doctor Love, a mad scientist, is working on a new hybrid virus which attacks both humans and computer programs written in functional languages. Her assistant, Frederick Thirst, has to do most of the computer programming while Doctor Love works mostly at the bio lab during alternate shifts. Between them, one of them usually sleeps on a sofa in the lab. During the course of the play, Thirst discovers the beauty of functional languages and convinces Doctor Love to abandon her evil plan. As the sun rises, they fall in love on the sofa. End of play.

Figure 1 shows the set groundplan. The director sets the following requirements for the lighting:

- The main *acting areas*—gravitational centers for the actors' movements—are on the chair behind the desk, the two lab tables, the sofa, and on the downstage center area between the desk and the lab tables.
- When Doctor Love emits her evil laugh, always behind the upstage lab table, she must look particularly evil.
- The contrast between the normal, sterile lighting during most of the play and the sunrise at the end must be very strong and visible.

A lighting designer, faced with these requirements, will usually start by lighting the basic acting areas: the chair behind the desk, the sofa and the two lab tables, and the center. Each such area will need several fixtures pointed at it to ensure good illumination and facial modelling, possibly with differing intensities. The designer will probably light the two lab tables separately. Additionally, she might place footramps under the downstage lab table for the Frankenstein-type evil-laugh lighting. A number of yellow-orange backlights provide the sunrise lighting.

These basic *lighting components* appear in a variety of combinations as lighting *looks* during the play: Basic, neutral lighting is a simple combination of the desk-area lighting, the sofa, the lab table lighting (itself consisting of two subcomponents), and the center area. Monologues might dim down all but one of these components to highlight a particular area. Special combinations include the evil-laugh look and the sunrise.

Technically, all of these lighting looks are merely intensity specifications for the stage fixtures—if the fixtures are numbered with indices, looks are simply vectors of intensities. However, it is immediately obvious that the intentions of the lighting designer carry a hierarchical structure.

*Design Goals.* Most commercial lighting control systems only support the construction of lighting components with depth two, and do not store their hierarchical structure at all. At present, only two consoles support hierarchical modelling for lighting looks, but their usage is discouraged by inflexible user interfaces, intricate semantic issues and insufficient documentation [3, 9]. The effect is that existing systems represent looks at the control or *implementation level* rather than at the conceptual level of the design. In fact, the schism between these two levels bears striking similarities to the difference between imperative and declarative programming.

Therefore, the major goal of Lula is a more faithful representation of the structure of a lighting design. This requires re-examining all basic design premises of existing systems, and has resulted in a complete redesign of the very concept of the lighting control system: Lula has been developed from scratch, both implementation-wise and conceptually.

Lula also tries to address another shortcoming of existing systems: These systems exhibit significant non-linearities between the user-interface controls and the actual situation on stage. Lighting designers often find themselves operating a control on the console and wondering, why nothing happens on stage, or why something different happened from what they expected. As a result, Lula's lighting component model is based on a rigorous formal specification. This specification is the basis for Lula's internal data representations, but, more importantly, determines the structure of its graphical user interface. The uniformity of the specification is not a guarantee, but a necessary prerequisite and good indicator for the usability of the interface.

*This Paper.* The faithful representation of the structure of lighting looks, described in Sections 2, 3, and 4 is the main innovation of Lula, and hence the main topic of this paper. Another central aspect of the system is its treatment of animated lighting, which builds upon Functional Reactive Programming, described in Section 5. The paper briefly reviews substrate considerations in Section 6, and experience gained in practice in Section 7.

## 2. BUILDING CUES

Consider the hierarchical structure of the lighting looks as suggested by the play as shown in Figure 2. The sofa, for example, has three fixtures pointing at it hooked to electrical channels 7, 14, and 23, at some specified intensity. Similarly, the ellipses under the other basic components stand for some combinations of electrical channels and the intensities of their fixtures. The "Lab tables" components has two subcomponents, each of which itself consists of subcomponents. The hierarchy builds up with the "Basic" components for the basic stage look, and the "Sunrise" look consisting of a dimmed version of "Basic" and added "Sun" lighting, and the "Evil Laugh" look which features the Frankenstein footramp and again dimmed basic lighting.

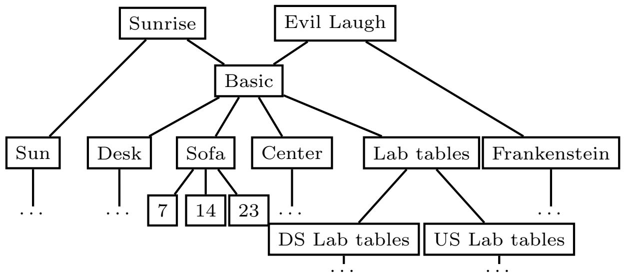


**Figure 2: Partial cue structure of *Doctor Love*.**

The resulting structure of the lighting components is a directed acyclic graph, and it is desirable that a lighting design system support manipulating such graphs. Lula calls static lighting components *cues* (a somewhat unfortunate term in retrospect, but it has stuck). The simplest cues represent single electrical channels. More complex cues consist of several subcues which might themselves have more structure. In lighting designs more realistic than the example presented here, the dag is often significantly deeper.

Note that cues correspond to *conceptual* entities in the lighting design rather than "implementation details" like what fixture performs what function or to which electrical circuit it is connected. Fixtures occur only at the leaves of the dag structure. Above the leaf level, the dag is independent of the concrete stage. This is an essential step forward from current commercial systems which ignore the structure and represent *only* the individual parameter settings

The example design above involves only one single, additive operator for cue composition: The fixtures involved in lighting the "Lab tables" cue are the union of the fixtures in "DS Lab tables" and "US Lab tables" at their respective intensities. If both component cues share a fixture, the compound cue has the fixture at the *maximum* of the intensities in the component cues, a principle called *Highest Takes Precedence* or *HTP* in the industry.

Two other combinators besides HTP have proven useful for cue construction in practice:

- Cue *subtraction* removes a set of fixtures from a cue. This is useful when talking about cues that result from other cues by taking parts away: "The entire stage without the upstage-left corner," for example.

- Cue *restriction* combines two component cues in a similar way as HTP does, but behaves differently for shared fixtures: one of the two cues has precedence for specifying the intensities of shared fixtures. This occurs in situations like "The entire stage, but with the kitchen table darker than the rest."

The cue language is effectively a *domain-specific language* (DSL) [22]. It has methodological similarities to Haskore [12] and recent work in using combinator languages to describe financial contracts [13].

## 2.1 Simple Cue Terms

Initially, it is easiest to consider a setting with exclusively theatrical fixtures which only have intensity control. However, the concepts presented here extend straightforwardly to multi-parameter fixtures. Section 4 shows how.

The basic algebra for cues builds on three primitive sorts: The *cue* sort is for cues themselves. *factor* represents a scalar factor; the scale function uniformly scales the intensity of a cue. Moreover, *fixture* represents a fixture, with an assumed constant for every fixture. Here is a signature for the operations on cues:

$$\begin{aligned}
&\text{fromfixture} : \textit{fixture} \to \textit{cue}\\
&\text{scale} : \textit{factor} \times \textit{cue} \to \textit{cue}\\
&\text{black} : \textit{cue}\\
&_\sqcup_ : \textit{cue} \times \textit{cue} \to \textit{cue}\\
&_/\!\!/_ : \textit{cue} \times \textit{cue} \to \textit{cue}\\
&_\setminus_ : \textit{cue} \times \textit{cue} \to \textit{cue}
\end{aligned}$$

The black cue denotes darkness. The scale function scales the intensity of a cue by some factor. (Presumably, the signature would also contain constructors for *factor* values. These were omitted for brevity.)

The *fromfixture* constructor converts a fixture into a cue containing only that fixture at maximum intensity. The combinators $\_\sqcup\_$, $\_/\!\!/\_$, and $\_\setminus\_$ are HTP, restriction, and cue difference, respectively.

## 2.2 Carrier Sets for Cues

Ultimately, the goal is to construct an equational specification for the cue operators in the previous section. Since the cue operators originate from the application domain where they already have fixed semantics, I first formalize the semantics, and backtrack from there to the specification.

A cue conceptually consists of a set of fixtures that are part of the cue, and intensities for these fixtures. The "cue contains fixture" notion is explicit: The model distinguishes between a cue $c$ which does not contain some fixture $f$ and a cue $c'$ which differs from $c$ only by including $f$ at zero intensity. The resulting algebra is called $A^0$.

An intensity is a non-negative real number, bounded by some maximum value $M$:

$$\mathbb{I} := \mathbb{R}^{0,+}_{\leq M}$$

Its carrier set $A^0_{\textit{fixture}}$ for *fixture* contains elements for all fixtures. $A^0_{\textit{cue}}$ is a set with:

$$A^0_{\textit{cue}} \subseteq \mathcal{P}(A^0_{\textit{fixture}}) \times (A^0_{\textit{fixture}} \rightsquigarrow \mathbb{I})$$

$A^0_{\textit{fixture}} \rightsquigarrow \mathbb{I}$ is the set of partial functions from $A^0_{\textit{fixture}}$ to $\mathbb{I}$. A cue must define intensities for exactly the fixtures it contains. Hence, $A^0_{\textit{cue}}$ is the largest set fulfilling the above condition as well as:

$$(F,p) \in A^0_{\textit{cue}} \iff F = \textit{dom}(p).$$

Factors are non-negative real numbers:

$$A^0_{\textit{factor}} := \mathbb{R}^{0,+}$$

## 2.3 Semantics of Cues

The next step in constructing $A^0$ is assigning meaning to its constants and operators. The black cue is easiest:

$$\text{black}^{A^0} := (\varnothing, \varnothing)$$

The fromfixture constructor assembles a cue from a single fixture at its maximum intensity:

$$\text{fromfixture}^{A^0}(f) := (\{f\}, \{f \mapsto M\})$$

The scale function scales all fixtures in a cue uniformly:

$$\text{scale}^{A^0}(\mu, (F,p)) := (F, p') \text{ where } p'(f) := \min(\mu \cdot p(f), M)$$

The HTP combinator merges the fixtures involved, and assigns maximal intensities:

$$(F_1,p_1) \sqcup^{A^0} (F_2,p_2) := (F_1 \cup F_2, p)$$
$$\text{where } p(f) := \begin{cases} p_1(f) & \text{for } f \notin F_2 \\ p_2(f) & \text{for } f \notin F_1 \\ \max(p_1(f), p_2(f)) & \text{otherwise} \end{cases}$$

Restriction also merges the fixtures involved, but gives precedence to the intensities of the second cue:

$$(F_1,p_1) /\!\!/^{A^0} (F_2,p_2) := (F_1 \cup F_2, p)$$
$$\text{where } p(f) := \begin{cases} p_1(f) & \text{for } f \notin F_2 \\ p_2(f) & \text{otherwise} \end{cases}$$

The difference combinator is the set-theoretic difference between the fixtures contained in the operand cue:

$$(F_1,p_1) \setminus^{A^0} (F_2,p_2) := (F_1 \setminus F_2, p_{1|F_1 \setminus F_2})$$

# 3. AXIOMS AND THEOREMS FOR CUES

The $A^0$ algebra has a number of pleasant properties which will form an axiomatic basis for the specification. The proofs of the axioms and theorems presented are straightforward and have been omitted for brevity [21]. Here are the three most immediate axioms[1]:

### 3.1 Axiom

*HTP is commutative and associative.*

### 3.2 Axiom

*HTP and restriction are idempotent. For every cue c:*

$$\begin{aligned} c \sqcup^{A^0} c &= c \\ c /\!\!/^{A^0} c &= c \end{aligned}$$

### 3.3 Axiom

*For any cue c:*

$$c \sqcup^{A^0} \text{black}^{A^0} = c \tag{1}$$
$$c /\!\!/^{A^0} \text{black}^{A^0} = c \tag{2}$$
$$\text{black} /\!\!/^{A^0} c = c \tag{3}$$
$$c \setminus^{A^0} \text{black}^{A^0} = c \tag{4}$$
$$\text{black}^{A^0} \setminus^{A^0} c = \text{black}^{A^0} \tag{5}$$
$$c \setminus^{A^0} c = \text{black}^{A^0} \tag{6}$$

[1] They are called axioms here because they are axioms in the resulting specification. At this point, they still require proofs of their validity in $A^0$. The theorems, in contrast, are exclusively derived from the axioms.

Restriction is expressible in terms of HTP and difference:

**3.4 Lemma**
*For any cues $a$ and $b$:*

$$a /\!\!/^{A^0} b = (a \setminus^{A^0} b) \sqcup^{A^0} b$$

Moreover, $\setminus$ and $\sqcup$ have some of the properties of their set-theoretic counterparts:

**3.5 Axiom**
*The following equations hold for all cues $a$, $b$, and $c$:*

$$\begin{aligned}
(a \sqcup^{A^0} b) \setminus^{A^0} c &= (a \setminus^{A^0} c) \sqcup^{A^0} (b \setminus^{A^0} c) && (7)\\
a \setminus^{A^0} (b \sqcup^{A^0} c) &= (a \setminus^{A^0} b) \setminus^{A^0} c && (8)\\
(a \setminus^{A^0} b) \setminus^{A^0} c &= (a \setminus^{A^0} c) \setminus^{A^0} b && (9)\\
(a \setminus^{A^0} b) \setminus^{A^0} c &= (a \setminus^{A^0} (b \setminus^{A^0} c)) \setminus^{A^0} c && (10)
\end{aligned}$$

**3.6 Theorem**
*Restriction is associative. For all cues $a$, $b$, and $c$:*

$$a /\!\!/^{A^0} (b /\!\!/^{A^0} c) = (a /\!\!/^{A^0} b) /\!\!/^{A^0} c$$

**3.7 Theorem**
*Restriction left-distributes over HTP. For cues $a$, $b$, $c$:*

$$(a \sqcup^{A^0} b) /\!\!/^{A^0} c = (a /\!\!/^{A^0} c) \sqcup^{A^0} (b /\!\!/^{A^0} c)$$

Note that restriction does *not* right-distribute over HTP.

Finally, a number of trivial axioms concern the interaction of scaling with the various cue combinators:

**3.8 Axiom**
*For any factor $\mu$ and cues $a$ and $b$:*

$$\begin{aligned}
\mathrm{scale}(\mu, \mathrm{black}) &= \mathrm{black} && (11)\\
\mathrm{scale}(\mu, a \sqcup b) &= \mathrm{scale}(\mu, a) \sqcup \mathrm{scale}(\mu, \mathrm{b}) && (12)\\
\mathrm{scale}(\mu, a /\!\!/ b) &= \mathrm{scale}(\mu, a) /\!\!/ \mathrm{scale}(\mu, \mathrm{b}) && (13)\\
\mathrm{scale}(\mu, a \setminus b) &= \mathrm{scale}(\mu, a) \setminus b && (14)
\end{aligned}$$

## 3.1 Differentiating Cues and Fixture Sets

The definition of cue difference is somewhat ugly: it implicitly disregards the intensity specifications of its second argument, treating it as a mere set of fixtures. Making the distinction between a cue and its fixture set explicit yields another cue operator: the *complement*. For a cue $c$, the complement yields a cue which contains exactly those fixtures not contained in $c$. Besides making the specification slightly more pleasant, this step also has practical bearing on the use of the specification for the construction of Lula's graphical user interface, as I will show later.

The new signature for cue terms includes an new sort *fixtureset* for fixture sets:

$$\begin{aligned}
&\mathrm{fromfixture} : \mathit{fixture} \to \mathit{cue}\\
&\mathrm{scale} : \mathit{factor} \times \mathit{cue} \to \mathit{cue}\\
&\mathrm{black} : \mathit{cue}\\
&_\sqcup_ : \mathit{cue} \times \mathit{cue} \to \mathit{cue}\\
&_/\!\!/_ : \mathit{cue} \times \mathit{cue} \to \mathit{cue}\\
&\downarrow _ : \mathit{cue} \to \mathit{fixtureset}\\
&\bar{_} : \mathit{fixtureset} \to \mathit{fixtureset}\\
&_ \setminus _ : \mathit{cue} \times \mathit{fixtureset} \to \mathit{cue}
\end{aligned}$$

The natural algebra for this signature, $A_1$, is a straightforward modification of $A_0$. Here are the differences between the two: First off, fixture sets are sets of fixtures:

$$A^1_{\mathit{fixtureset}} := \mathcal{P}(A^1_{\mathit{fixture}})$$

Cue abstraction extracts the first component from a cue:

$$\downarrow^{A^0} (F, p) := F$$

The complement has the set-theoretical interpretation:

$$\overline{F} := A^1_{\mathit{fixture}} \setminus F$$

Thus, subtracting the complement of $c$, $\overline{c}$, works like applying a stencil.

Double complement is the identity on fixture sets:

**3.9 Axiom**
*For any fixture set $F$, the following holds:*

$$\overline{\overline{F}} = F$$

The semantics of the difference operator needs to reflect the change in signature:

$$(F_1, p_1) \setminus^{A^1} F_2 := (F_1 \setminus F_2, p_{1|F_1 \setminus F_2})$$

To avoid overly complicating the presentation and having to rewrite all terms involving differences, cue abstraction will sometimes be implicit from here on. With this notational agreement, all axioms of Section 3 hold as before. In $A^1$, another axiom holds:

**3.10 Axiom**
*For cues $a$, $b$, and $c$, the following equation holds:*

$$a \setminus^{A^1} (b \setminus^{A^1} c) = (a \setminus^{A^1} b) \sqcup A^1 (a \setminus^{A^1} \overline{c})$$

Actually, it would be easier to make statements about difference if the intersection operator from set theory were available. However, intersection does not have clear intuitive meaning when applied to lighting. Section 3.2 contains more discussion of this issue.

## 3.2 A Domain-Theoretic Interpretation of Cues

Denotational semantics uses partial orderings on the elements of semantic domains to distinguish the amount of information they contain [10]. From a conceptual standpoint, lighting works in a similar way: the more light there is on stage, the more is visible to the audience.

Consequently, it is possible to define an ordering on cues, induced by the HTP operator:

$$a \sqsubseteq b :\Longleftrightarrow a \sqcup b = b$$

The meaning of the $\sqsubseteq$ operator becomes clearer when related to the semantics:

$$\begin{aligned}
&(F_1, p_1) \sqsubseteq^{A^1} (F_2, p_2) :\Longleftrightarrow\\
&\qquad F_1 \subseteq F_2 \text{ and } p_1(f) \leq p_2(f) \text{ for all } f \in F_1
\end{aligned}$$

Thus, $a \sqsubseteq b$ means that all fixtures contained in $a$ are also contained in $b$ and shine at least as bright as in $b$. Therefore, $a \sqsubseteq b$ is pronounced "$a$ is at least as dark as $b$" or "$b$ is at least as bright as $a$."

**3.11 Theorem**
*$\sqsubseteq$ is a partial order.*

This makes $\sqcup$ a least upper bound. It is possible to drive the analogy between cues and semantic domains even further:

**3.12 Theorem**
*Cues form a complete partial order in $A^1$: every $\omega$-chain in $A^1_{cue}$ has a least upper bound.*

Here is another straightforward correspondence of the cue algebra with semantic domains:

**3.13 Theorem**
*HTP and restriction are continuous in both arguments.*

The relationship with domains ends with the difference operator which is continuous in its first but not in its second argument. This is hardly surprising as, conceptually, difference *removes* information from a cue.

Note also that fixture sets are a natural abstraction of cues in a domain-theoretic sense—cues and fixture sets form a Galois connection [10].

The domain-theoretic interpretation of cues has no great relevance in the intensity-only setting. However, it is a crucial ingredient in formulating the extension to multi-parameter fixtures.

# 4. MULTI-PARAMETER FIXTURES

The equational specification so far is surprisingly specific to fixtures which allow intensity control only. Non-intensity parameters require a more elaborate treatment, both in the implementation and in the formal specification. There are two reasons for this:

- Intensity is qualitatively different from other parameters: If the intensity is zero, no change to any of the other parameters is visible. On the other hand, *every* change in intensity is visible, at least in principle.
- Even though most numeric parameters do have a limited parameter range, they mostly do not have an evident ordering. (Is a color $a$ with a higher proportion of red than another color $b$ larger or smaller?)

Applying the semantic view to a lighting installation with multi-parameter light helps find a principle for resolving the problem. This principle translates fairly directly into a new algebraic specification for cues.

The domain-theoretic interpretation of cues assumes that, for any two cues $a$ and $b$, a least upper bound $a \sqcup b$ exists: $a \sqcup b$ is a cue which reveals everything $a$ and $b$ reveal. $a \sqcup b$ is at least as bright as both $a$ and $b$.

This view is not powerful enough for handling multi-parameter fixtures: Even though every fixture in a cue $a$ might be brighter than every fixture in cue $b$, the fixtures might point in different directions, therefore revealing some other thing entirely, and leaving the target of $a$ in the dark.

The specification of different settings for a non-intensity parameter in the same cue term is called a *conflict*. Such cue terms do not correspond to well-defined parameter settings. Consequently, two cues $a$ and $b$ containing multi-parameter fixtures can only be comparable if the non-intensity parameters of all fixtures included in $a$ and $b$ have the same settings. This means that not all pairs of cues have least upper bounds. Furthermore, not all pairs of cues have HTPs: $a \sqcup b$ does not exist if $a$ and $b$ share fixtures with different settings for non-intensity parameters.

Thus, the domain-theoretic interpretation yields a precise notion of conflicts. This is in sharp contrast with existing lighting control systems which do not notify the user of conflicts at all, and only offer ad-hoc mechanisms for resolving them. In contrast, Lula informs the user when she tries to introduce a conflict into the cue hierarchy, and gives (based on the specification below) detailed information about the source of the conflict.

In the absence of conflicts, all results from the intensity-only case carry over to multi-parameter fixtures.

## 4.1 Modelling Parameter Transformations

Besides the issue of conflict, the introduction of multi-parameter fixtures requires generalizing the notion of scaling to arbitrary *transformations* of parameters. Intensity scaling stands for a function from intensity values to intensity values. Similarly, all transformations represent function from parameter values to parameter values. Here are some examples beside intensity scaling:

**Color Set** A color-set transformation sets the color of all fixtures in a cue that allow color control.

**Pan/Tilt Set** A pan/tilt-set transformation sets the pan and tilt parameters of all moving lights of a cue.

**X/Y/Z Set** An X/Y/Z-set transformation sets stage coordinates for the moving lights of a cue to focus on.

**X/Y/Z Offset** An X/Y/Z-set transformation moves the light beams of moving lights by an offset in the horizontal plane at the specified Z coordinate. This is useful, for example, to correct light positioning on dancers with a preprogrammed choreography.

Each transformation is specific to a certain parameter, and applies uniformly to all the fixtures in a cue.

As the operator assembles cues by applying transformations and applying the cue combinators, transformations accumulate in two different ways:

**Composition** arises when a transformed cue is transformed again: the two transformations compose functionally.

**Juxtaposition** arises from the HTP combination of two cues which contain a common fixture. For two intensity transformations, juxtaposition produces their least upper bound. For non-intensity parameters, juxtaposition is only meaningful in the absence of conflicts.

The introduction of these two concepts justifies separating out the specification of transformations into their own specification. The next subsection shows how composition and juxtaposition interact with the cue combination operators.

The signature for transformations only supports parameters for intensity and pan/tilt. However, adding further parameters is completely analogous. The signature differentiates between sorts for specific transformations for intensity and pan/tilt—*itrafo* and *pttrafo* and general transformations *trafo*.

The $\circ$ operators are composition operators for transformations of individual parameters. $\|$ is for juxtaposition of intensity transformations. A transformation in *trafo* is conceptually a tuple of an intensity transformation and a pan/tilt transformation: the # operator assembles one from its components. The $\diamond$ operator composes two transformations; $\star$ is for juxtaposition.

$$
\begin{aligned}
&\text{scale} : \mathit{factor} \to \mathit{itrafo}\\
&\epsilon_{\mathit{itrafo}} : \mathit{itrafo}\\
&\_ \circ_{\mathit{itrafo}} \_ : \mathit{itrafo} \times \mathit{itrafo} \to \mathit{itrafo}\\
&\_ \parallel \_ : \mathit{itrafo} \times \mathit{itrafo} \to \mathit{itrafo}\\
&\text{pan/tilt} : \mathit{angle} \times \mathit{angle} \to \mathit{pttrafo}\\
&\epsilon_{\mathit{pttrafo}} : \mathit{pttrafo}\\
&\_ \circ_{\mathit{pttrafo}} \_ : \mathit{pttrafo} \times \mathit{pttrafo} \to \mathit{pttrafo}\\
&\_\#\_ : \mathit{itrafo} \times \mathit{pttrafo} \to \mathit{trafo}\\
&\_ \diamond \_ : \mathit{trafo} \times \mathit{trafo} \to \mathit{trafo}\\
&\_ \star \_ : \mathit{trafo} \times \mathit{trafo} \to \mathit{trafo}
\end{aligned}
$$

Juxtaposition is conceptually a partial function; hence, it uses a special exception value notrafo in the trafo sort *trafo*.

Presumably, the scale constructor builds intensity-scale transformations from *scale* values, pan/tilt constructs transformations that set pan/tilt from two *angle* values. Moreover, $\epsilon_{\mathit{itrafo}}$ and $\epsilon_{\mathit{pttrafo}}$ are special constants meaning "no intensity (or no pan/tilt, respectively) transformation specified."

Transformations obey the following laws:

$$
\begin{aligned}
&\epsilon_{\mathit{itrafo}} \circ_{\mathit{itrafo}} i = i\\
&i \circ_{\mathit{itrafo}} \epsilon_{\mathit{itrafo}} = i\\
&\epsilon_{\mathit{pttrafo}} \circ_{\mathit{pttrafo}} p = p\\
&p \circ_{\mathit{pttrafo}} \epsilon_{\mathit{pttrafo}} = p\\
&(i_1\#p_1) \diamond (i_2\#p_2) = (i_1 \circ_{\mathit{itrafo}} i_2)\#(p_1 \circ_{\mathit{pttrafo}} p_2)\\
&\epsilon_{\mathit{itrafo}} \parallel i = i\\
&i_1 \parallel i_2 = i_2 \parallel i_1\\
&(i_1\#\epsilon_{\mathit{pttrafo}}) \star (i_2\#p) = (i_1 \parallel i_2)\#p\\
&t_1 \star t_2 = t_2 \star t_1\\
&p_1 \neq \epsilon_{\mathit{pttrafo}}, p_2 \neq \epsilon_{\mathit{pttrafo}} \Longrightarrow (i_1\#p_1) \star (i_1\#p_2) = \text{notrafo}
\end{aligned}
$$

The propagation of the notrafo exception value is implicit.

It may seem strange that the specification does not allow composing two identity pan/tilt transformations. This restriction reflects Lula's user interface which distinguishes between the absence of a transformation and the presence of an identity transformation. Otherwise, the user could create a conflict simply by operating the slider associated with a pan/tilt transformation, which is not desirable.

Finding an algebra $A^2$ for this equational specification is straightforward: transformations are functions with special values for the $\epsilon$ constructors added.

Intensity transformations are mappings from intensities to intensities plus a bottom value. The $\epsilon$ constructors correspond semantically to bottom values, hence the symbols chosen for $A^2$:

$$
\begin{aligned}
A^2_{\mathit{itrafo}} &:= (\mathbb{I} \to \mathbb{I}) + \{\bot_{\mathit{pttrafo}}\}\\
\epsilon_{\mathit{itrafo}} &:= \bot_{\mathit{itrafo}}
\end{aligned}
$$

$A^2$ uses distinguished values for the $\epsilon$ constructors rather than the identity on the associated parameters. This in turn is to reflect the conflicts of the specification in the model which also is the basis for conflict detection in the implementation in Lula.

The scale function applies an intensity transformation in $A^1$ receives a new meaning in $A^2$: it turns a factor into a function which scales intensity values:

$$\text{scale}^{A^2}(\mu)(i) := \min(\mu \cdot i, M)$$

The definition of $A^2_{\mathit{pttrafo}}$ is analogous to that of $A^2_{\mathit{itrafo}}$: A pan/tilt setting consists of two angles.

$$
\begin{aligned}
A^2_{\mathit{angle}} &:= \mathbb{R}^{0,+}_{\leq 2\pi}\\
A^2_{\mathit{pttrafo}} &:= ((A^2_{\mathit{angle}} \times A^2_{\mathit{angle}}) \to (A^2_{\mathit{angle}} \times A^2_{\mathit{angle}})) + \{\bot_{\mathit{pttrafo}}\}\\
\epsilon_{\mathit{pttrafo}} &:= \bot_{\mathit{pttrafo}}
\end{aligned}
$$

The pan/tilt operator constructs a constant function:

$$\text{pan/tilt}^{A^2}(a_p, a_t) := \widehat{(a_p, a_t)}$$

For non-bottom intensity transformations $i_1$ and $i_2$ or pan/tilt transformations $p_1$ and $p_2$, composition is simply functional composition:

$$
\begin{aligned}
i_1 \circ^{A^2}_{\mathit{itrafo}} i_2 &:= i_1 \circ i_2\\
p_1 \circ^{A^2}_{\mathit{pttrafo}} p_2 &:= p_1 \circ p_2
\end{aligned}
$$

As an aside, note that, even if scaling is the only transformation available for intensities, it is not possible to represent a scaling transformation by its factor, and thus achieve composition of two intensity-scaling transformation by multiplication of their factors. To see why, consider the cue term:

$$\text{apply}(\text{scale}(0.5), \text{apply}(\text{scale}(2), \text{fromfixture}(f)))$$

Composition by factor multiplication would pleasantly reduce this to:

$$\text{apply}(\text{scale}(1), \text{fromfixture}(f))$$

and, hence, fromfixture$(f)$. Unfortunately, this is wrong: There is no such thing as "double maximum intensity" for a real fixture. Hence, apply(scale(2), fromfixture$(f)$) is equivalent to fromfixture$(f)$ in a faithful model. Compositionality really does require that the example term has $f$ only at half the maximum intensity.

To get back to defining compositions for intensity and pan/tilt transformations, the two bottoms are neutral with respect to composition as in the specification:

$$
\begin{aligned}
i \circ^{A^2}_{\mathit{itrafo}} \bot_{\mathit{itrafo}} &:= i\\
\bot_{\mathit{itrafo}} \circ^{A^2}_{\mathit{itrafo}} i &:= i\\
p \circ^{A^2}_{\mathit{itrafo}} \bot_{\mathit{pttrafo}} &:= p\\
\bot_{\mathit{itrafo}} \circ^{A^2}_{\mathit{pttrafo}} p &:= p
\end{aligned}
$$

Intensity transformations allow juxtaposition via forming the least upper bound:

$$(i_1 \parallel i_2)(v) := \max(i_1(v), i_2(v))$$

Finally, a transformation really is a tuple of an intensity and a pan/tilt transformation. Also, $\mathit{trafo}^{A^2}$ contains an exception element called $\;_{\mathit{trafo}}$:

$$\begin{aligned} A^2_{\mathit{trafo}} &:= (\mathit{itrafo} \times \mathit{pttrafo}) + \{\;_{\mathit{trafo}}\} \\ \mathrm{notrafo}^{A^2} &:= \;_{\mathit{trafo}} \end{aligned}$$

Composition of two transformation works by pointwise composition and must take care to preserve exceptions:

$$\begin{aligned} (i_1, p_1) \diamond^{A^2} (i_2, p_2) &:= (i_1 \circ^{A^2}_{\mathit{itrafo}} i_2, p_1 \circ^{A^2}_{\mathit{pttrafo}} p_2) \\ \;_{\mathit{trafo}} \diamond^{A^2} t &:= \;_{\mathit{trafo}} \\ t \diamond^{A^2} \;_{\mathit{trafo}} &:= \;_{\mathit{trafo}} \end{aligned}$$

Juxtaposition also works as in the specification:

$$\begin{aligned} (i_1, \bot_{\mathit{pttrafo}}) \star^{A^2} (i_1, p) &:= (i_1 \parallel^{A^2} i_2, p) \\ (i_1, p) \star^{A^2} (i_1, \bot_{\mathit{pttrafo}}) &:= (i_1 \parallel^{A^2} i_2, p) \\ (i_1, p_1) \star^{A^2} (i_1, p_2) &:= \;_{\mathit{trafo}} \quad \text{for } p_1, p_2 \neq \bot_{\mathit{pttrafo}} \\ \;_{\mathit{trafo}} \star^{A^2} t &:= \;_{\mathit{trafo}} \\ t \diamond^{A^2} \star \;_{\mathit{trafo}} &:= \;_{\mathit{trafo}} \end{aligned}$$

## 4.2 Modelling Multi-Parameter Cues

The new signature for cues with pan/tilt fixtures is an extension of the signature for transformations. The parts not related to the application of an intensity scale are largely unchanged.

Dealing with conflicts requires considerably more elaboration on the semantics of cues on the part of the specification:a conflict happens at the level of a parameter setting for a single fixture, so the specification needs to define how cues define parameter settings.To this end, the signature has a new sort *setting*. A *setting* is an association of a fixture with a transformation.

In the new signature, at setting has, for a fixture $f$ and a transformation $t$, the form $f@t$, pronounced "$f$ is at $t$."

The $\hookrightarrow$ operator relates a cue $c$ with a setting $s$: $c \hookrightarrow s$ means that $c$ specifies a setting $s$. The pronunciation of $c \hookrightarrow f@t$ is "$c$ has $f$ at $t$." Here is the signature:

$$\begin{aligned} &\_@\_ : \mathit{fixture} \times \mathit{trafo} \to \mathit{setting} \\ &\_ \hookrightarrow \_ : \mathit{cue} \times \mathit{setting} \to \mathit{bool} \\ &\mathrm{fromfixture} : \mathit{fixture} \to \mathit{cue} \\ &\mathrm{apply} : \mathit{trafo} \times \mathit{cue} \to \mathit{cue} \\ &\mathrm{black} : \mathit{cue} \\ &\_\sqcup\_ : \mathit{cue} \times \mathit{cue} \to \mathit{cue} \\ &\_/\!\!/\_ : \mathit{cue} \times \mathit{cue} \to \mathit{cue} \\ &\downarrow\_ : \mathit{cue} \to \mathit{fixtureset} \\ &\bar{\_} : \mathit{fixtureset} \to \mathit{fixtureset} \\ &\_ \setminus \_ : \mathit{cue} \times \mathit{fixtureset} \to \mathit{cue} \end{aligned}$$

Here is an equational specification for the new operators:

$$\begin{aligned} &\mathrm{apply}(t, \mathrm{black}) = \mathrm{black} \\ &\mathrm{apply}(t, a \sqcup b) = \mathrm{apply}(t, a) \sqcup \mathrm{apply(t, b)} \\ &\mathrm{apply}(t, a /\!\!/ b) = \mathrm{apply}(t, a) /\!\!/ \mathrm{apply(t, b)} \\ &\mathrm{apply}(t, a \setminus b) = \mathrm{apply}(t, a) \setminus b \\ &\mathrm{apply}(t_1, \mathrm{apply}(t_2, c)) = \mathrm{apply}(t_1 \diamond t_2, c) \\ &\mathrm{apply}(t, \mathrm{fromfixture}(f)) \hookrightarrow f@t \\ &a \hookrightarrow f@t_1 \wedge b \hookrightarrow f@t_2 \Longrightarrow (a \sqcup b) \hookrightarrow f@(t_1 \star t_2) \\ &a \hookrightarrow f@t_1 \wedge \neg(\exists t_2 . b \hookrightarrow f@t_2) \Longrightarrow (a \sqcup b) \hookrightarrow f@t_1 \\ &b \hookrightarrow f@t \Longrightarrow (a /\!\!/ b) \hookrightarrow f@t \\ &a \hookrightarrow f@t_1 \wedge \neg(\exists t_2 . b \hookrightarrow f@t_2) \Longrightarrow (a /\!\!/ b) \hookrightarrow f@t_1 \\ &a \hookrightarrow f@t_1 \wedge \neg(\exists t_2 . b \hookrightarrow f@t_2) \Longrightarrow (a \setminus b) \hookrightarrow f@t_1 \end{aligned}$$

The rules for apply look much like the rules for scale in the old specification. However, there is an additional rule explaining the composition of transformations in terms of composition of its components.

The rules for apply are able to propagate transformations to the leaves of a cue term, the fromfixture terms. Moreover, the composition rule for apply allows squashing several nested transformations into one.

In turn, the $\hookrightarrow$ relation infers an obvious setting for a fixture from a leaf term of the form $\mathrm{apply}(t, \mathrm{fromfixture}(f))$. The other rules propagate settings upwards inside compound cues. This upwards propagation works only through the regular cue combinators, not through transformation applications. Hence, inferring setting information for the fixtures contained in a cue means first pushing the transformations inwards, squashing them there and inferring setting information for the fixture leaf nodes, and then propagating the settings back outwards.

Building an algebra $A^2$ for the specification is more involved than the construction of $A^1$. First off, $A^2$ includes $A^1$ unchanged. The construction of the *cue* carrier must map fixtures to transformations instead of intensities as in $A^1$. A well-defined cue must only have defined transformation. An exceptional transformation, when part of a cue, produces an exceptional cue. The new $A^2_{\mathit{cue}}$ is a set with:

$$A^2_{\mathit{cue}} \subseteq (\mathcal{P}(A^2_{\mathit{fixture}}) \times (A^2_{\mathit{fixture}} \rightsquigarrow (A^2_{\mathit{trafo}} \setminus \{\;_{\mathit{trafo}}\})) + \{\;_{\mathit{cue}}\}$$

As above, $A^2_{\mathit{cue}}$ must also fulfill the following condition:

$$(F, p) \in A^2_{\mathit{cue}} \Longleftrightarrow F = \mathit{dom}(p).$$

The black cue has the same meaning as before:

$$\mathrm{black}^{A^2} := (\varnothing, \varnothing)$$

A single-fixture cue has only an undefined transformation associated with it:

$$\mathrm{fromfixture}^{A^2}(f) := (\{f\}, \{f \mapsto (\bot_{\mathit{itrafo}}, \bot_{\mathit{pttrafo}})\})$$

The setting constructor @ is simple tupling:

$$\begin{aligned} A^2_{\mathit{setting}} &:= A^2_{\mathit{trafo}} \\ f@t &:= (f, t) \end{aligned}$$

Application of a transformation treats all fixtures contained in a cue uniformly:

$$\mathrm{apply}^{A^2}(t, (F, p)) := (F, p') \text{ with } p'(f) := t \diamond^{A^2} p(f)$$

Of the cue combinators, HTP is the most interesting as it involves the juxtaposition of transformation, and, therefore, the potential for conflicts:

$$(F_1, p_1) \sqcup^{A^2} (F_2, p_2) := (F_1 \cup F_2, p)$$
$$\text{where } p(f) := \begin{cases} p_1(f) & \text{for } f \notin F_2 \\ p_2(f) & \text{for } f \notin F_1 \\ p_1(f) \star^{A^2} p_2(f) & \text{otherwise} \end{cases}$$

This definition is only valid if all $p_1(f) \star^{A^2} p_2(f)$ involved do not produce an exception. If juxtaposition does produce a transformation exception, the HTP is undefined, signaling a conflict:

$$(F_1, p_1) \sqcup^{A^2} (F_2, p_2) := \ {}_{cue}$$
$$\text{if there is an } f \in F_1 \cap F_2 \text{ with } p_1(f) \star^{A^2} p_2(f) = \ {}_{trafo}$$

Restriction and difference basically work as before:

$$(F_1, p_1) /\!\!/^{A^0} (F_2, p_2) := (F_1 \cup F_2, p)$$
$$\text{where } p(f) := \begin{cases} p_1(f) & \text{for } f \notin F_2 \\ p_2(f) & \text{otherwise} \end{cases}$$

$$(F_1, p_1) \setminus^{A^0} (F_1, p_1) := (F_1 \setminus F_2, p_{|F_1 \setminus F_2})$$

Relating settings to cues is straightforward in $A^2$:

$$(F, p) \hookrightarrow^{A^2} (f, t) :\iff f \in F \text{ and } p(f) = t$$

**4.1 Theorem**
*$A^2$ is a model of the equational specification.*

## 4.3 A Graphical User Interface for Cues

Naturally, ordinary users prefer not to deal with algebraic constructs. Therefore, Lula offers a simple graphical user interface for constructing and editing cues. This is made possible by another algebraic result about the equivalence of cue terms to terms with bounded height:

**4.2 Definition**
*Assume constants $c_1, \ldots, c_n$ : cue. An atom is either one of the $c_i$ or a term of the form* fromfixture$(f)$ *for a fixture constant $f$.*

*A flat cue term has the following form:*

$$\bigsqcup_{i=1}^{n} s_i$$

*where each of the $s_i$ is either an atom or has one of the following two forms:*

$$a_1 /\!\!/ a_2 /\!\!/ \ldots /\!\!/ a_{n_i} \text{ with } a_1, \ldots, a_{n_i} \text{ atoms}$$

$$a_1 \setminus \widehat{a_2} \cdots \setminus \widehat{a_{n_i}} \text{ with } a_1, \ldots, a_{n_i} \text{ atoms}$$

*Each $\widehat{a_i}$ is either $a_i$ itself or its complement $\overline{a_i}$. The difference operator is assumed to be left-associative.*

The fundamental result associated with flat cue form is that every cue term has one:

**4.3 Theorem**
*Again assume has constants $c_1, \ldots, c_n$ : cue. For each cue term $t$, there exists a flat cue term $t'$ equivalent to $t$. Moreover, it is possible to choose $t'$ such that it contains the same atoms as $t$.*

**Proof** By term rewriting, using the axioms and theorems of the previous sections. □

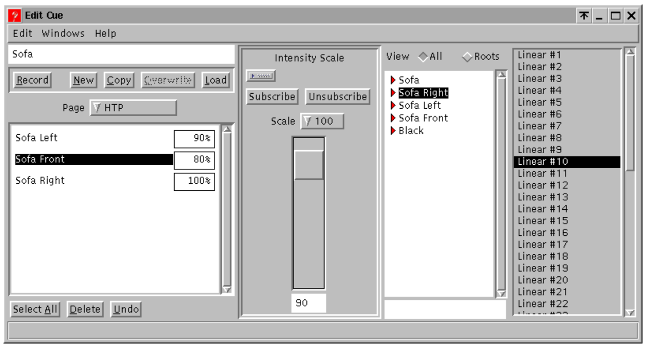


**Figure 3: Cue editor.**

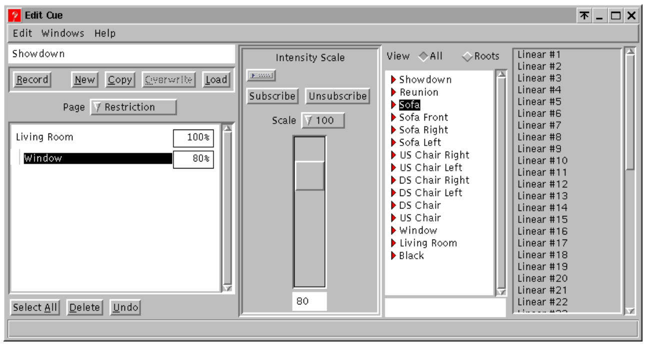


**Figure 4: Cue editor displaying a restriction page.**

Flat cue form, along with other properties such as associativity or commutativity, corresponds directly to desirable properties of a graphical user interface. Hence, the design of Lula's editor widget for cues is based directly upon cue flat form. Figure 3 shows such a cue editor widget. In Lula, each cue has a name (`Sofa` in this case, taken from yet another play) and the user constructs new cues in flat cue form where each atom in the sense of Definition 4.2 is either a fixture or another named cue together with a set of transformations.

The white area on the left represents an $s_i$ subterm, a so-called *page* in the terminology of Lula: it is a list box showing the atomic subcues. There are three kinds of pages, `HTP`, `Restriction`, and `Difference`. The latter two correspond directly to the form of the $s_i$ subterms in Definition 4.2; `HTP` pages represent terms of the form $a_1 \sqcup a_2 \sqcup \ldots \sqcup a_{n_i}$ This is in trivial accordance with Theorem 4.3. These terms have their own page type because HTP is the most common form of cue combination.

The list box on the right displays available fixtures, the white box to its left displays a list of already-defined cues; it is a tree widget, and the user can expand the hierarchical display by clicking on the little triangles. The cue editor shows an HTP page with three subcues at different intensities. The slider in the middle changes the intensities of selected subcues.

Figure 4 shows a cue editor displaying a restriction page. Here, the first cue has a special role. All subsequent cues may have a complement operator applied to it: the user can click on the little box to the left of the `Window` cue to toggle the complement.

## 5. ANIMATING LIGHTING

While cues are the cornerstones of lighting design, they do not cover the time element in a show: The lighting design for a theatrical performance usually consists of a sequence of cues with transitions (so-called *fades*) between them. However, the real animated lighting extends far beyond fades between cues: even in theater, the lighting might have to change dynamically, concerts often involve animated multi-parameter fixtures with impressive moving effects.

Lula internally expresses all light changes as animations in term of *Functional Reactive Programming* or *FRP*. Functional Reactive Programming is a programming technique for representing values that change over time and react to events. It is suitable for a wide range of applications, among them graphics animations as pioneered by Elliott's *Fran* system [2], graphical user interfaces [19], and robotics [16]. As it turns out, it is also applicable to animated lighting.

For constructing complex animations, the user has direct access to FRP via a built-in domain-specific programming language called *Lulal*, a restricted dialect of Scheme. Lulal allows the construction of reusable components for animations which are accessible through a simpler graphical user interface, enabling even non-programmers to design their own animations. This section builds on the terminology of FRP [2].

### 5.1 Presets

Lighting animations build upon cues as their static components. Procedurally, the user constructs a library of cues with the graphical user interface presented in the previous section, and then uses these as the building blocks for animated lighting. A lighting animation can create looks that do not correspond to any cue the user has created: many animation operators, such as fades, have no counterpart in the cue language. Moreover, offering the HTP operator in the animation language is dangerous, as it may lead to unpredictable conflicts at inopportune times. Therefore, the primitive static entity in Lulal is not the cue, but the *preset*. Just like a cue, it specifies fixture parameter settings, and every cue is also a preset. Presets differ from cues in the set of operations available for their construction.

The key ingredient for the representation of animated lighting is the *preset behavior*, a specialized representation for presets changing over time. Lulal offers a rich algebra for constructing preset behaviors, as well as traditional restriction and difference operators on presets, but no HTP.

### 5.2 The Lulal Language

Lulal is a higher-order, purely functional, strongly typed language with parametric polymorphism. Lulal's syntax is mostly borrowed from Scheme [14]. As its semantics also builds upon Scheme, the description of Lulal in the section is brief and focuses on the differences.

The design of Lulal makes a number of departures from Scheme syntax and semantics which gear Lulal more specifically towards its use in an end-user application. Most of them are restrictions on Scheme semantics to prevent an average user from making unnecessary mistakes. Here is a summary of the changes:

- No side effects.
- Explicit recursion is not allowed.
- Lulal is strongly typed. Its type system is a modified version of the popular Hindley/Milner system [1]. This restricts expressiveness somewhat, but catches many common programming errors before the actual show.
- The language allows a limited form of overloading of constant values with constant behaviors.
- Lulal has additional syntax for dealing with the task monad, similar to Haskell's `do` notation [11].
- Cues are predefined objects in Lulal, represented by a string literal containing its name.

Lulal's value domain includes behaviors, events, and tasks for constructing reactive values. To simplify the work of the programmer, Lulal allows a limited form of overloading: a value of a base time is also overloaded as the corresponding constant behavior. This results in a programming style similar to programming Fran animations in Haskell [2].

Tasks represent actions to be done in the reactive framework: a task defines a preset behavior as well as a time at which it ends.

Tasks form a *monad* [23], a special value domain for representing computations. Lulal's use of monads for representing tasks is similar to the one in used in Monadic Robotics [16]. Lulal supports the usual monadic combinators `return` and `bind` as well as a Haskell-`do`-like `sequence` operator. Within the monad, the Lulal semantics propagate two values: the start time of a task's action as well as the preset defined by the previous action at its end.

*FRP Primitives.* Lulal's basic facilities for FRP are similar to Fran's: it has lifted versions of the standard numerical operations, integral and derivative behaviors, a primitive `time` behavior, and time transformation.

Moreover, the user has access to the usual FRP event algebra with primitive event constructors from alarm times and GUI components, as well as the usual event-handling combinators, the usual `switcher` and `stepper` procedures for converting events into behaviors, as well as event merging.

*Preset Behaviors.* In an animated setting, presets generalize naturally to *preset behaviors*. In Lulal, cues define the primitive preset behaviors. A string literal is implicitly a reference to a cue. The behavior associated with a cue changes its value whenever the user modifies the cue. Lulal contains a subset of the primitives available for cue construction: `restrict-with` is the lifted restriction operator, and `subtract` is the lifted difference.

*Transformation Behaviors.* A Lulal program can obtain new lighting animations by applying transformation behaviors to preset behaviors. A number of constructors for such transformations are available:

**Intensity** Lulal treats behaviors of reals as intensity transformation behaviors.

**Pan/tilt** The `pan/tilt` constructor builds a pan/tilt transformation behavior from a tuple of two behaviors of reals, specifying the pan and tilt angles, respectively. It is also possible to construct a pan/tilt transformation behavior from three Cartesian coordinate behaviors via a constructor called `xyz`. `Xyz-offset` creates a behavior which shifts the target of a light beam in a specified horizontal plane by two behaviors of the X and Y coordinates.

**Color** A number of constructors are available for making color transformation behaviors from behaviors specifying the RGB, HSV, or CMY components.

*Applying Transformations.* A number of combinators create new preset behaviors by combining transformation behaviors with preset behaviors: the result is a preset behavior resulting from the application of the values of the transformation behavior to those of the preset behaviors. `Scale` scales the intensity of a preset behavior by a real behavior. `With-pan/tilt` applies the pan/tilt transformations resulting from a pan/tilt transformation behavior; `with-color` does the same with a color transformation behavior.

## 5.3 Tasks

Tasks are the top-level entities in Lulal relevant to the user: When the user specifies an action that triggers an animation, she must specify a task defined by the Lulal program which describes the animation.

A task defines a preset behavior as well as an event whose first occurrence signals completion of the action of the task. The user can combine arbitrary preset behaviors with events to form tasks. In addition, Lulal provides fades as primitive tasks. The user can combine tasks by sequencing or running the actions side-by-side.

## 5.4 Example

Consider the following lighting assignment: A moving light follows an actor on stage. Another one follows the first one with a two-second delay. Another one follows with a four-second delay.

Assume that some sensory equipment is hooked up to the console which reports the actor's position. Further assume the procedure `get-position` returns a behavior tuple which reports the X, Y, and Z coordinates of the actor's head (or whatever portion of his body should be lit). Here is an expression yielding an appropriate task:

```
(let* ((position (get-position))
       (later-position
         (time-transform position
                         (delayed time -2.0)))
       (even-later-position
         (time-transform later-position
                         (delayed time -2.0))))
  (preset-behavior->task
    (restrict-with
      (with-pan/tilt (xyz position)  "Followspot #1")
      (restrict-with
        (with-pan/tilt (xyz later-position)
                       "Followspot #2")
        (with-pan/tilt (xyz even-later-position)
                       "Followspot #3")))))
```

`Time` is the primitive time behavior. `Delayed` time-transforms a behavior by shifting it. `Preset-behavior->task` directly turns a preset behavior into a task that never ends.

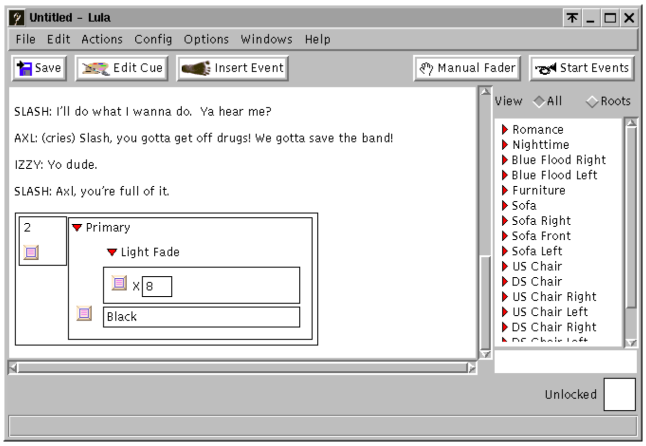


**Figure 5: Script editor showing final light fade.**

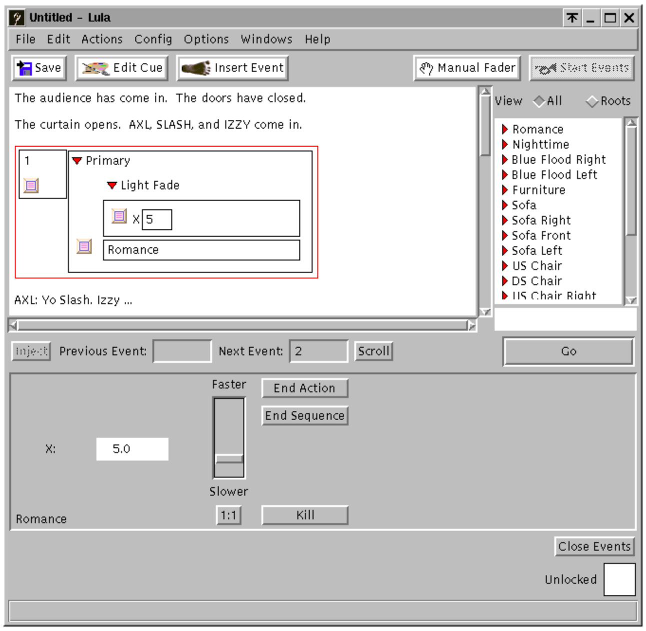


**Figure 6: Start of playback.**

## 5.5 Assembling a Show

FRP (or, as a matter of fact, any kind of programming) is not for every user of a lighting control system. Thus, it is crucial to make the flexibility afforded by Lulal available to these user and to the designer of dynamic lighting components, but hide it from the user who merely wants to assemble a show from cues, fades and prefabricated pieces.

Figure 5 shows Lula's *script editor*. It is essentially a simple multimedia editor (based directly on powerful components already provided by the PLT Scheme system) which allows pasting *events* into a script. An event corresponds to an event on stage which requires a coordinated lighting change.

Figure 5 shows the final lighting event of a simple show, a fade-to-black which takes 8 seconds. Each event can triggle multiple lighting changes simultaneously, arranged sequentially or in parallel.

Since the multimedia editor allows including ordinary text and images, it can, for example, hold the playscript of a theatrical production. This has proved to be tremendous advantage as compared with traditional systems, where lighting events carry numbers which the operator needs to coordinate with numbers written in a paper version of the script.

Figure 6 shows the beginning of *playback* during a show: the script editor splits to show the progress of the lighting events in the bottom half. Lula offers numerous opportunities for manual intervention: the operator can suspend lighting changes, interrupt and abort actions as well as slow down or speed up the animation. Most of the GUI controls are directly hooked up to FRP components; this has made the implementation extremely simple.

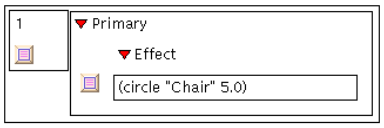


**Figure 7: Effect event in script window.**

Figure 7 shows how the user can access programmed animations written in Lulal: she simply specifies a term which yields a lighting task.

# 6. SUBSTRATE CONSIDERATIONS

Lula is written in Scheme and runs top PLT Scheme [4]. The feature set of PLT Scheme makes it suitable for application development in general and for Lula in particular:

- concurrent threads of execution,
- a GUI framework portable between X Windows, Microsoft Windows, and MacOS [7, 6] which includes a collection of GUI widgets for multimedia editors,
- a higher-order, fully-parameterized module system [5],
- an object system supporting parametric inheritance via *mixins* [8],
- the *Zodiac* framework [15] for building scanners and parsers for Scheme-like languages.

In particular, multithreading and the portable GUI framework are enabling technologies for applications like Lula.

This combination is still fairly rare in functional-language implementations. If functional languages are to succeed in the marketplace, more implementations must provide this kind of application-level support.

Two aspects of PLT Scheme have proved somewhat more problematic for Lula:

- PLT Scheme uses a non-incremental garbage collector. Machines have only recently become fast enough make GC pauses short enough to not cause noticeable delays during complex lighting animations. (However, straightforward linear animations runs just fine on a 133-Mhz Pentium.)
- Threads are not particularly lightweight—each thread takes up about 20kBytes. This precludes some programming styles that depend on threads being extremely cheap (such as the use CML-style synchronous operations [18]), and some effort was necessary to keep the number of threads down, particularly in the sampling subsystem and in the user interface.

All in all, however, PLT Scheme has proven an excellent substrate for the development of Lula.

# 7. LULA IN PRACTICE

Lula has been in development since 1997. It has since been in constant use in the University Theater, and has a number of users there. It has toured with Theater U34, a local theater outfit, to a number of venues in Tübingen, Reutlingen, Stuttgart, and Munich. It has also been in use at Stuttgart State Theater by the lighting designers there.

In theatrical use, Lula drastically cuts down on the time usually needed for programming the control system, often by a factor of two or more. This is significant since the time spent on programming the control system is usually not available for set construction or rehearsal. Moreover, allows the lighting designer or director to communicate the structure of the lighting design prior to the on-stage phase. The time saved immediately translates to better designs.

Lula is especially effective for touring productions: Since it allows separating the conceptual components of a design from its implementation, the operator can preserve large parts of the programming between venues. In this environment, the use of Lula can dramatically improve the lighting because the time available on-stage is usually very limited.

The response from users has been uniformly positive. The University Theater has seen a successful bootstrap of the system and conducts its own workshops on the use of the system, taught by operators rather than computer programmers. This takes about two hours on the average for groups of 4–12; after that, the participants are on their own, and usually do well. Results are especially good when we teach lighting design in conjunction with the use of the system, as the principles of both go hand in hand.

Perhaps surprisingly, experienced lighting operators find it harder to get used to Lula than beginners, mainly because of their long exposure to traditional lighting control system and the resulting assumptions about how "a lighting console works." They initially try to use Lula as they use a traditional system, and are disappointed when they see no significant immediate improvement. These designers need to see Lula applied to an existing design to see its benefits.

The animated lighting component of Lula is still under development. We expect similar results in that arena.

# 8. CONCLUSION

Lula is a powerful system for lighting design and control. Its main departure from existing systems is its modelling of the conceptual structure of a lighting design rather than its implementation. This simplifies input and editing of lighting designs, and greatly improves the flexibility of the result as compared with existing systems.

Both design and the implementation have benefited from the use of advanced software engineering techniques:

- The use of a functional wide-spectrum language has greatly shortened development time. The first prototype of Lula (presented at CeBIT '97) was finished in just under a week.
- The rigorous algebraic model for cues and its domain-theoretic interpretations have been instrumental in creating a consistent and powerful user interface for creating and editing cues.
- The embedded DSL approach, pioneered in the functional programming community is pervasive in Lula, both in the design of the cue algebra and the animation subsystem.
- Functional Reactive Programming is an ideal technique for expressing lighting animation.

The outward design of Lula mirrors its internal structure.

Consequently, Lula demonstrates that modern functional language substrates are eminently suitable for application development.

*Availability.* A demo of the current stable Lula system is available under `http://www-pu.informatik.uni-tuebingen.de/lula/`.

*Acknowledgments.* I would like to thank the anonymous reviewers for their excellent comments which were instrumental in revising the paper.

## 9. REFERENCES

[1] Damas, L., and Milner, R. Principal type-schemes for functional programs. In *Proc. 9th Annual ACM Symposium on Principles of Programming Languages* (1982), ACM, pp. 207–212.

[2] Elliott, C., and Hudak, P. Functional reactive animation. In *Proc. International Conference on Functional Programming 1997* (Amsterdam, The Netherlands, June 1997), M. Tofte, Ed., ACM Press, New York.

[3] ETC, Inc. *Obsession II User Manual*, version 4 ed. Middleton, Wisconsin, 1998. Available electronically as `http://www.etcconnect.com/user_manuals/consoles/obsn_4.pdf`.

[4] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. The DrScheme project: An overview. *SIGPLAN Notices 33*, 6 (June 1998), 17–23.

[5] Flatt, M., and Felleisen, M. Units: Cool modules for HOT languages. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Montreal, Canada, June 1998), K. D. Cooper, Ed., ACM, pp. 236–248. Volume 33(5) of SIGPLAN Notices.

[6] Flatt, M., and Findler, R. B. *PLT Framework: GUI Application Framework*. Rice University, University of Utah, Aug. 2000. Version 103.

[7] Flatt, M., and Findler, R. B. *PLT MrEd: Graphical Toolbox Manual*. Rice University, University of Utah, Aug. 2000. Version 103.

[8] Flatt, M., Krishnamurthi, S., and Felleisen, M. Classes and mixins. In *Proc. 25th Annual ACM Symposium on Principles of Programming Languages* (San Diego, CA, USA, Jan. 1998), L. Cardelli, Ed., ACM Press, pp. 171–183.

[9] Flying Pig Systems Ltd. *WHOLEHOG II Handbook*, version 3.2 ed. London, UK, 1999. Available electronically as `http://www.flyingpig.com/Hog2/cgi/ftp.cgi?file=pub/nils/hogIIv3_2.pdf`.

[10] Gunter, C. A. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, Cambridge, MA, 1992.

[11] Haskell 98, a non-strict, purely functional language. `http://www.haskell.org/definition`, Dec. 1998.

[12] Hudak, P., Makucevich, T., Gadde, S., and Whong, B. Haskore music notation — an algebra of music —. *Journal of Functional Programming 6*, 3 (May 1996), 465–483.

[13] Jones, S. P., Eber, J.-M., and Seward, J. Composing contracts: An adventure in financial engineering. In Wadler [24], pp. 280–292.

[14] Kelsey, R., Clinger, W., and Rees, J. Revised$^5$ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11*, 1 (1998), 7–105. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[15] Krishnamurthi, S. Zodiac: A framework for building interactive programming tools. Tech. Rep. Technical Report CS TR 95-262, Rice University, Department of Computer Science, 1995.

[16] Peterson, J., Hager, G., and Hudak, P. A language for declarative robotic programming. In *Proceedings of the International Conference on Robotics and Automation Information 1999* (Detroit, Michigan, May 1999), Y. F. Zheng, Ed., IEEE Press.

[17] Reid, F. *The Stage Lighting Handbook*, third ed. A & C Black, London, England, 1987.

[18] Reppy, J. H. Synchronous operations as first-class values. In *Proc. Conference on Programming Language Design and Implementation '88* (Atlanta, Georgia, July 1988), ACM, pp. 250–259.

[19] Sage, M. FranTk — a declarative GUI language for Haskell. In Wadler [24], pp. 106–117.

[20] Shelley, S. L. *A Practical Guide to Stage Lighting*. Focal Press, Oxford, England, 1999.

[21] Sperber, M. *Computer-Assisted Lighting Design and Control*. PhD thesis, Universität Tübingen, 2001.

[22] van Deursen, A., Klint, P., and Visser, J. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices 35*, 6 (June 2000), 26–36.

[23] Wadler, P. Monads for functional programming. In *Advanced Functional Programming*, vol. 925 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1995, pp. 24–52.

[24] Wadler, P., Ed. *International Conference on Functional Programming* (Montreal, Canada, Sept. 2000), ACM Press, New York.