Computer-Assisted Lighting Design and Control

Dissertation
der Fakultät für Informatik
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Michael Sperber
aus Marburg/Lahn

Tübingen
2001

Tag der mündlichen Qualifikation: 9. Mai 2001
Dekan: Prof. Dr. Andreas Zell
1. Berichterstatter: Prof. Dr. Herbert Klaeren
2. Berichterstatter: Prof. Dr. Wolfgang Straßer

Abstract

This dissertation shows that computer-based lighting control systems can support the lighting design process considerably better than traditional consoles. It describes the Lula Project, a new software package for lighting design and control, that implements this level of support. Lula's focus is on the conceptual ideas behind a lighting design rather than the concrete lighting fixtures used to put it on stage. Among the innovative aspects of the system are its model for designing static lighting looks and its subsystem for programmable continuous animated lighting. Lula's application design is centered around the idea of *componential lighting design* that allows the user to express a lighting design as a hierarchy of components. Lula is a result of the rigorous application of high-level software engineering techniques and implementation technology from the general realm of functional programming. The high-level structure of the application rests upon stratified design, algebraic modelling, and domain-specific languages. Among the implementation techniques instrumental to Lula are automatic memory management, higher-order programming, functional data structures, data-directed programming, parametric inheritance, and concurrent programming.

Zusammenfassung

Computer-basierte Systeme für Beleuchtungssteuerung sind in der Lage, den Lichtdesigner weitaus besser zu unterstützen als es derzeit marktübliche Steuerkonsolen tun. Das Thema dieser Dissertation ist ein solches System, das Projekt Lula. Lula ist eine neue Software für Lichtregie und Beleuchtungssteuerung, welche die Modellierung der konzeptuellen Elemente eines Lichtdesigns ermöglicht, unabhängig von der konkreten Realisierung auf der Bühne. Unter den innovativen Aspekten des Systems ist das Modell für den Entwurf statischer Beleuchtungsszenen sowie das Subsystem für programmierbare, stetig animierte Beleuchtung. Das übergeordnete Prinzip bei Lula ist *komponentenbasierte Lichtregie*, die es dem Benutzer erlaubt, ein Lichtdesign als eine Hierarchie von Komponenten auszudrücken. Lula ist das Resultat konsequenter Anwendung von Entwurfs- und Implementierungs-Techniken aus dem Bereich der funktionalen Programmierung. Die High-Level-Struktur des Systems baut auf stratifiziertes Design, algebraische Modellierung und anwendungsspezifische Programmiersprachen. Unter den Implementationstechniken, die entscheidend bei der Entwicklung von Lula waren, befinden sich automatische Speicherverwaltung, Higher-Order-Programmierung, funktionale Datenstrukturen, datengesteuerte Programmierung, parametrische Vererbung und nebenläufige Programmierung.

Contents

Acknowledgements

So Sailor, our histories have been
somewhat intertwined.
—Lula in *Wild at Heart*

I have not done it alone. While the coding and internal aspects of the Lula project have rested solely on my own shoulders, a number of people have made significant contributions to the project; without them, it would not be where it is today.

First of all, my thanks go to my boss and advisor, Prof. Herbert Klaeren, who agreed to the initial idea and provided the material means to support its ongoing development. Most importantly, he humored my enthusiasm for the project, sanctioned the time I put into it, and has stood by it despite Lula's so far limited success in the commercial arena. Thanks also go to Peter Thiemann for early encouragement, and to Prof. Wolfgang Straßer for agreeing to co-review this dissertation.

A lighting console, however fancy its functionality, becomes a worthless piece of scrap metal once it crashes. A number of people helped test Lula and helped me bring its bug count down to production quality. Before release, Ea Wolfer, Till Grab, and Sabine Ferber provided invaluable services. Also, a trek of lighting designers at the University's Brecht-Bau theater suffered through the successive releases, prominent among them Henry Toma, again Ea Wolfer, Sven Göttner, and Mark Zipperlein. Their patience and tolerance for the snags of the system has been truly heroic. Also, I thank the countless actresses and actors as well as the audience members unwittingly turned beta testers.

I thank Eckart Steffens of Soundlight, Hannover, for valuable feedback. He also provided me with free samples of the DMX512 hardware his company makes, and kept me posted on the DMX512/2000 standardization effort.

Werner Dreher provided tremendous help during the development of Lula 2000, Lula's first hardware interface. What I didn't know but should have known about digital circuit design could fill a book, and without Werner, the lights would not have come up at CeBIT '97. Thanks also go to Johannes Hirche for designing the layout for the Lula DMX.

I have received truly great support from the members of PLT at Rice, providing instant response on any problem, bug report or suggestion I submitted, no matter how half-baked or moronic: Matthew Flatt and Robby Findler, the developers-in-chief, and to Shriram Krishnamurthi and Matthias Felleisen for further support.

The help of Till Grab of the Depot at the State Theater in Stuttgart was crucial in the design of Lula's user interface. Till also commented on drafts of some of the chapters in this dissertation, providing important feedback and corrections. Any remaining errors are my sole responsibility. Thanks also go to Peter Müller and Pit Schmidt for pointing me in Till's direction as well as helpful comments. The members of the `Lichttechniker` mailing list, Michael Doepke and Hado Hein in particular, have provided stimulating discussion and encouragement.

For whatever is readable in this work, my father bears much responsibility; he taught me how to write. From Ann Carvalho, I learned proper English many years

ago. The remaining errors, oversights and other deficiencies are mine. There will no doubt be too many of them to count.

Many other people have provided feedback, direct or indirect, on Lula, or supported my work in other ways. There are just too many to list them completely and accurately. A collective thanks goes to them all.

Part I

Introduction

Hello... Who is this?...
... Sailor Ripley... Can I talk to Lula?
—Sailor in *Wild at Heart*

Chapter 1

The Lula Project

Hey, my snakeskin jacket ... Thanks,
baby ... Did I ever tell you that
this here jacket for me is a symbol
of my individuality and my belief
in personal freedom?
—Sailor in *Wild at Heart*

Lula is a computer-based system for lighting design and control. Its main focus is theatrical production, but it also eminently suitable for lighting dance, musical concerts, and industrial presentations.

Lula is a practical result of research in software engineering. Hence, its contributions fall in two groups:

- The application of advanced software engineering methods leads to good software *design*. In this case, it has led to a lighting system superior to commercially available consoles.
- The application of of advanced software engineering methods leads to rapid and reliable software *construction*.

This introduction gives an overview of Lula, both as a lighting control system and as a software project. The former describes Lula from a lighting designer's perspective, whereas the latter will take on a software engineer's viewpoint. A short history of the project and an overview of the dissertation follow.

1.1 Lula as a Lighting Control System

A computer-based lighting console can support effective lighting design, drastically save time during the design process, and boost creativity. The key to maximum effectiveness of a lighting console is how close its user interface is to the thought process of the lighting designer: a console with good conceptual modelling capabilities can represent the design as it emerges in the mind of the designer.

Unfortunately, the conceptual modelling capabilities of existing consoles, even extensively engineered high-end models, are not very well developed. Instead, these consoles still view a lighting installation as a flat collection of fixture[1] parameter settings. As the operator of a lighting console creates the programming for a show, she frequently needs to deal with low-level details which have no bearing on the

[1] A *fixture* is a general term for a source of light on stage. Chapter 2 contains an extensive discussion of fixtures.

lighting itself—the specific cabling path used to connect a fixture to the console, channel assignments on multi-function lights, and how all of this relates to console controls. In fact, modern computer-backed consoles still fundamentally operate at the same level as the manual consoles of the 1970s.

Lula is a computer-based lighting system with a special focus on the design process rather than the implementation details of a particular lighting installation. It is substantially more effective than other consoles available on the market today. Here are some of Lula's technological highlights:

- Lula runs on ordinary computers, most notably PCs. This makes the system easily implementable and extensible. Lula can cooperate with a wide range of interfaces to connect to the electrical lighting systems of modern stages. Specifically, it has full support for the ubiquitous digital protocol for controlling lighting, DMX512 [DMX512-1990, 1990, DINDMX, 1997].

- Lula operates at the conceptual level of a lighting design. The central idea and chief innovation of Lula is *componential lighting design* which views a lighting installation as a hierarchy of interdependent components. Componential lighting design reduces the complexity of operating the user interface by and order of magnitude, and significantly eases creating a design as well as making changes to it at a later time.

- A corollary to componential lighting design is the separation between the concepts of a design and its actual implementation on a concrete stage. This capability, called *venue independence*, allows touring productions to preserve most of their light programming as they move from one stage to the next. As time is usually very limited for setting up a stage, this is actually an enabling technology: many designs are sufficiently complicated that conventional consoles will simply not allow finishing the programming in the time available.

- Lula has extensive support for *animated light*. Rather than treating animated light as mere effects, Lula sees animated light as a continuous phenomenon. This makes Lula suitable for creating light choreography, a feat extraordinarily difficult with conventional consoles.

- Lula's playback capabilities are based on a *script*, rather than the "cue lists" of conventional consoles which are just numbered sequences of action. A Lula script is actually a multimedia document, and besides offering advanced capabilities for sequential and parallel playback, it allows storing *all* playback information associated with a show; the operator need not refer to external "track sheets" on paper.

As a result of Lula's high-level modelling capabilities, the lighting designer can use it during most of the design process, implementing ideas as they evolve, rather than after the fact as is mostly the case with conventional consoles.

1.2 Lula as a Software Project

The application of advanced software engineering methods and of functional programming language technology yields reliable and maintainable software an order of magnitude more quickly than the methods currently common in industry.

As a software project, the implementation of a lighting control system poses a number of challenges to software designers and implementors:

- Designing the lighting for any full-length production is a complex process. The software must present a user interface which helps manage that complexity. Internally, it must be able to represent the complexity of the design.

- During the show itself, the lighting console functions live and in real time. Moreover, as shows sometimes do not go quite as planned, the console must offer extensive controls for live intervention.
- Light on the stage is one of the indispensable prerequisites for almost any show (one person on stage and one in the audience being the others). Hence, the software *must* function reliably.

In the particular case of Lula, it also had to be done very quickly. At the time of writing, about six man-months have been available for the Lula project, one of which was spent on hardware design and construction.

The techniques and technologies which were instrumental in bringing Lula to life address two levels of the development process: *structural modelling* and *implementation*. Three important foundations for the modelling part were the following:

Algebraic modelling Algebraic modelling uses algebraic techniques like equational reasoning and abstract data types to define a semantic foundation for the data the program deals with. Lula's model for the design of static lighting looks is based on careful algebraic design which has a well-defined semantics.

Domain-specific languages Domain-specific languages help structure large applications into a language capable of handling the problem domain conceptually, and programs in that language that solve actual problems from the domain. In the case of Lula, domain-specific languages operate at several levels: the cue algebra forms a small domain-specific language. Moreover, Lula employs an embedded domain-specific language to express lighting animations. It also provides a specialized stand-alone language called *Lulal* to the user of the system which allows her to design her own animations.

Stratified design Lula's core subsystems are organized as a stack of linguistic levels building on one another: the cue system builds on the system for parameter settings, the embedded domain-specific animation language builds on the cue system, and Lulal builds upon the embedded language.

The Lula code base depends on a number of advanced implementation techniques:

- automatic memory management
- higher-order-programming
- functional programming
- data-directed programming
- parametric inheritance
- concurrent programming

This combination is presently only available in advanced functional programming languages [Thiemann, 1994, Hudak, 2000, Field and Harrison, 1988, Hughes, 1990]. Lula in particular is written in Scheme [Kelsey *et al.*, 1998], a particularly flexible functional imperative language. It makes extensive use of the added features of the DrScheme programming language environment [Felleisen *et al.*, 1998, Flatt *et al.*, 1999].

Even though functional programming is still something of a novelty in industrial development, the techniques cited above are really the folklore of the functional programming community. Hence, Lula's implementation is everything but rocket science. However, as most work in functional programming still happens in the research community, comparatively few complete end-user applications exist.

The rigorous use of formal modelling techniques and functional programming has a deep impact on the development process has consequences far beyond shorter time-to-market, improved reliability and lower maintenance costs.

The development of Lula has shunned some of the more conventional trends in software engineering, in particular the use of object-oriented design, modelling languages, and CASE tools. Moreover, Lula uses object-oriented programming quite rarely, even though the environment it runs in has extensive facilities for it.

There is another aspect to Lula's design, namely the application of modern user-interface design techniques which play an important part in making Lula the effective application it is. These techniques, however, are standard by now, and the novelty is mainly in their application to an application domain which has ignored them in the past. Hence, the particulars of Lula's user interface construction are not explicitly covered in this dissertation.

1.3 A Brief History of the Lula Project

In late 1996, Prof. Herbert Klaeren's working group for programming languages and compiler construction was preparing a presentation for the 1997 CeBIT computer trade show. Our work is ordinarily not very well suited for such presentations, since programming language research is by definition work necessary *before* a computer-science-related product goes into production. Moreover, it is difficult to demonstrate in 10 minutes the significance of an innovation which shows its benefits primarily in large long-term projects. So we needed a demo.

One of my spare time interests is theater and specifically theater lighting. For productions I directed, I usually insisted on doing the lighting design as well.

I had long been disappointed by the fact that although modern lighting installations have computer-driven operating consoles, these consoles operate on principles which have not progressed very far since the 70s. As a consequence, creating the lighting for a show always takes too long, the results are often less than satisfying.

More specifically, it also always took *me* too long, and much of the time I was able to spend on creating a lighting design was spent operating the console or waiting for someone else to do so. The ideas I had in my head about what a lighting design should look like and what its structure is never seemed to translate very well into what I had to tell the console. On several occasions, I have actually had changes which should have been trivial to tell the console about delay the opening of a show.

I had long wanted to invest some time into the development of a new type of console. The 1997 CeBIT proved to be a perfect opportunity: thus the first version of Lula was born. Back then, the hardest part of getting Lula was interfacing a computer to the electrical installation. I, having no prior experience in hardware design and construction, got out my books on digital circuit design from the early 1980s. I hand-soldered the *Lula 2000*, a bulky logic design, in about four weeks, starting five weeks before the CeBIT. That left one week for the software, just right for demonstrating that the use of functional programming could actually help create a complete application.

Thus Lula 1 premiered at the 1997 CeBIT. It already featured the fundamental concepts of componential lighting design which form the cornerstone of the Lula in use today.

After the CeBIT, Lula moved to the Brecht-Bau Theater, the University stage, and the theater groups there have made extensive use of it in a large number of productions. The original Lula 2000 also remains in use there to this day.

In late 1997 and 1998, Lula 2 was created, a more polished version of the original Lula with largely unchanged functionality. Special emphasis was put into making the program reliable—a crash on a lighting console is somewhat more serious than

with other applications, and can be a very valid reason to reject a console and favor another, even though it might be technically inferior.

In early 1999, an effort began to make Lula into a potential marketable product. On the outset, this meant making Lula compatible with DMX512, the digital protocol used on most major stages. Another hardware interface, the *Lula DMX*, was the result. With this new addition, Lula toured extensively with U34, a local theater outfit and thus received extensive testing in practice.

At about the same time, commercial DMX512 interfaces became affordable. As a consequence, Lula's driver structure now supports many vendors of such interfaces. Lula 3 was a major rewrite of many parts of the system. Besides many additional bells and whistles, it was the first Lula to allow the creation of a lighting *script* which obviates the tedious back-and-forth looking between a cue book and the computer screen. Lula 3 was also more in tune with modern graphical-user interfaced standards, and also was the first Lula to have a written manual as well as a tutorial.

Lula 3 is the currently distributed version of Lula; used correctly, it is the most effective console for theater lighting available.

When Lula 3 came out, many lighting designers who had tested it encouraged me to extend Lula to also support concert lighting which is different from theater lighting in many respects. Specifically, they requested support for *moving lights* which is significantly harder than just doing theatrical lighting. Moreover, concert lighting requires some sort of „effect engine“ for the dynamic lighting common at larger concerts.

So I went back to the drawing board for yet another rewrite. On my own, I had concluded that the componential lighting idea had an underlying algebraic structure which I wanted to explore. Also, in 1997, I had visited Conal Elliott at Microsoft Research in Seattle who back then was working on a new declarative programming model for reactive animation, by now dubbed *Functional Reactive Programming*. By the time I started the work on Lula 4, I was pretty sure I could apply the same ideas and programming techniques to put a programmable effect engine into Lula. This happened, and Functional Reactive Programming is now actually responsible for *all* light changes in Lula.

At the time of writing of this dissertation, Lula 4 is still in the prototype phase, but well on its way to also become a finished production-quality application.

1.4 Contributions

To summarize, the main contributions of this dissertation are the following:

- This dissertation, to my knowledge, is the first systematic investigation of computer-assisted lighting design and control. Prior work in this area was mainly by the design departments of the various console manufacturers. However, as I argue in this dissertation, their results have been rather ad-hoc. Grab's excellent investigation of requirements lighting control systems is limited to theatrical lighting [Grab, 1995].

- Lula is the first lighting control system to support the novel idea of *componential lighting design*.

- This dissertation contains the first formalization of lighting look construction. Lula's cue model for componential look design is new and superior to that of available consoles.

- Lula is the first system to apply reactive animation to the domain of animated light. Lulal, Lula's domain-specific language for lighting animation, is the first

such language. Moreover, Lula is probably the first end-user application of functional reactive programming.

- The Lula application itself is an indicator that functional programming languages are excellently suited for industrial-strength application development.
- The extremely rapid development time needed for implementing Lula shows that modern functional programming language technology and programming paradigms scale well to large-scale applications.
- Lula shows some possible avenues for improvement of programming language technology to support application development even better.

1.5 Overview

The dissertation is structured in seven parts, this introduction being the first. Part II is a study of the application domain. The first chapter in this part discusses the nature of the most important hardware in lighting—the light sources themselves. Chapter 3 discusses common protocols for connecting fixtures to control electronics. Chapter 4 is a very brief introduction to the basics of lighting design, especially as they relate to requirements for lighting control software. Chapter 5 reviews some of the more advanced commercially available lighting control systems.

Part III offers a brief tour of the Lula software from the user's standpoint. Chapter 6 addresses the basic concepts of the system; Chapter 7 covers advanced issues, including animated light.

The architecture of Lula is the subject of part IV. The first chapter in that part discusses tools an techniques instrumental in the implementation of the system. Chapter 9 gives a structural overview of the system viewed as a reactive network, and the implementation technology used to connect its pieces.

Part V is concerned with Lula's support for designing static lighting "looks" from components. Chapter 10 establishes some requirements and prerequisites for successful modelling of looks. Chapter 11 is an extensive formal account of Lula's answer to the look construction question—its *cue* subsystem. This chapter also contains some implementation notes.

Animated lighting is covered in Part VI. Chapter 12 contains an extensive account of *functional reactive programming*, the implementation technology underlying the light animation. Chapter 13 covers the application of functional reactive programming to the application domain of lighting.

Part VII offers some closing remark. Chapter 14 assesses the success of the system design and its implementation. Chapter 15 contains on outlook on future work.

Part II

Lighting As We Know It

I just think about things as they
come up. I never been much of a planner.
—Lula in *Wild at Heart*

The application software designer does well in studying the application domain thoroughly. The domain of stage lighting has many facets among which software is only a small part. On the other hand, the software controlling the lighting for a show is at the nexus of a number of processes, each of which requires specific attention from the software designer. Some of these processes are technical, some of them creative in nature, and lighting control software must support all of them to be effective.

Moreover, electronic lighting control systems have existed for a long time. Hence, the design of a new one must build on the experience gained from the use and evaluation of previous systems. This is especially important in a field as tradition-infused as stage plays, concerts and other live performances: the practitioners of stage lighting are conservative in their adoptions of supposedly "new" methods—and justly so: the methods in current use have been sufficient to create wonderful, astonishing designs, and any new system must first prove that it can do what others can do, and more.

Lighting a show has three basic interrelating aspects:

Design An idea of what the stage lighting should look like.

Hardware The light sources; the mechanical and electrical systems supporting their operation.

Control The instructions given to the hardware to perform to their purpose; the electrical or electronic systems which communicate the instructions.

This part is an overview of these basic ingredients of lighting design. In order to be coherent to some degree, it touches upon a number of issues which are pertinent to Lula, but whose realization in Lula is outside the scope of this dissertation. The part starts off with the physical and technical: Chapter 2 describes the types of light sources in use on contemporary stages as well as the electrical prerequisites for making them work. Chapter 3 describe the protocols in use to communicate control instructions to the electrical installation.

After the hardware, the design process is the subject of Chapter 4 which gives a brief and necessarily incomplete account of some of its methods. Finally, Chapter 5 describes the control systems common in the industry today, and what they do to support practical lighting design and operation.

Chapter 2

Stage Fixtures

I love it when your eyes get wild,
honey. They light up all blue almost
and little white parachutes pop out
of 'em.
—Lula in *Wild at Heart*

A *fixture* is a source of light on the stage—the fundamental prerequisite for stage lighting. (Other words for fixtures are *luminaire* or *instrument*, or, specifically for stage fixtures, *lantern*.) As lighting designers need complete control over all aspects of lighting on stage, stage fixtures have a number of controllable parameters. This includes, beside basic visibility and brightness, many other aspects such as direction, distribution, color, focus, selectivity, beam shape, and so on. The lighting industry has produced a stunning variety of different fixture types to cater to the demands of the field. Modern multi-parameter light give remote control over many of these aspects to the lighting console; a lighting control system needs to provide some degree of modelling of such fixtures. For this reason, this chapter gives a basic overview of the most common types of stage fixtures as a basis for lighting control system design.

2.1 Parameters

A fixture consists of a lamp inside a housing. A stage fixture also has an optical system consisting of mirrors and lenses, as well as a rigging attachment, typically a yoke. This is only the most basic arrangement.

Stage fixtures allow control over a large number of qualities of the light produced. Such a quality is called a *parameter*. Most parameters are "static," referring to a quality of the light constant over a period of time. Here is a list of the basic static parameters:

Intensity or *brightness*. Changing the voltage applied to the lamp or adjusting an iris or lamelle changes the intensity.

Color Color is an inherent quality of the lamp (possibly intensity-dependent), and can be influenced by using colored *gels* (or *filters*) to block out parts of the light spectrum. A particularly important aspect of light from the viewpoint of the lighting designer is *saturation*, a particular aspect of color.

Direction Stage light is generally directional light, and always "points" in a certain direction defined by the position of the yoke,

Beam size Light leaves a fixture at a certain angle, possibly asymmetrically with respect to the direction of the beam. The exit angle defines the so-called *spread* of the beam. Moreover, the size and shape of the aperture in front of the lamp also has an effect on beam size.

Beam shape A beam of light can have a shape different from a circle, or even a pattern.

Beam focus Focus refers to the wavefront convergence of the light rays constituting the beam. On stage, focus is visible as a property of the edge of the light beam which might be soft or harsh. Focus imbues a corresponding quality on the objects it hits.

These are the static parameters. Some qualities of the light arise only indirectly from the combination of several parameters such as *distribution*, *flow* and *selectivity*. Moreover, the light of some fixtures have inherently dynamic qualities, most notably the light flashes produced by strobes.

Depending on the fixture type, the operator may only have control over some of the parameters. Moreover, the control may be *manual*: someone has to get up on a ladder and adjust the parameter, and it will stay the same until the next adjustment. Some controls are under *remote control*—the operator can adjust them from the control system. Most fixture types allow remote control via a *dimmer* (see below). Advanced, so-called *multi-parameter fixtures*, allow remote control over most if not all parameters of the light beam.

2.2 Dimmers

The most important remote control for any stage fixture is its intensity. A device able to gradually change the intensity of a lamp is called a *dimmer*.

With incandescent lamps, changing the applied voltage also varies the output. In particular, modern dimmers work by chopping off parts of the output waveform.

Nowadays, dimmers mostly come in packs integrating a number of output circuits. Remote control is either by a small input voltage or current per circuit, or via a digital protocol (see Chapter 3).

2.3 Lamps

Household lamps are rarely suitable for use on stage. They do not have near enough the required output power. Moreover, the frequent changes in intensity shorten the household lamp's life cycle to the point where it is no longer practical to maintain a large number of them in an average stage situation.

Three basic methods for generating light are common in stage fixtures:

Tungsten sources Most stage lamps use a tungsten filament to emit the light, just like household lamps. However, almost all tungsten stage lamps are *halogen lamps*: the bulb is filled with a halogen which binds to evaporating tungsten atoms, which eventually returns them to the filament. This greatly increases the life cycle as compared to a vacuum bulb. It also reduces light and color temperature in the output. Note that the relationship between input voltage and the various output parameters is not linear. Figure 2.1 shows a diagram of a section from a typical output distribution.

Discharge sources Discharge lamps contain two electrodes and an inert gas or vapor. An electric arc between the electrodes generates the light. Discharge

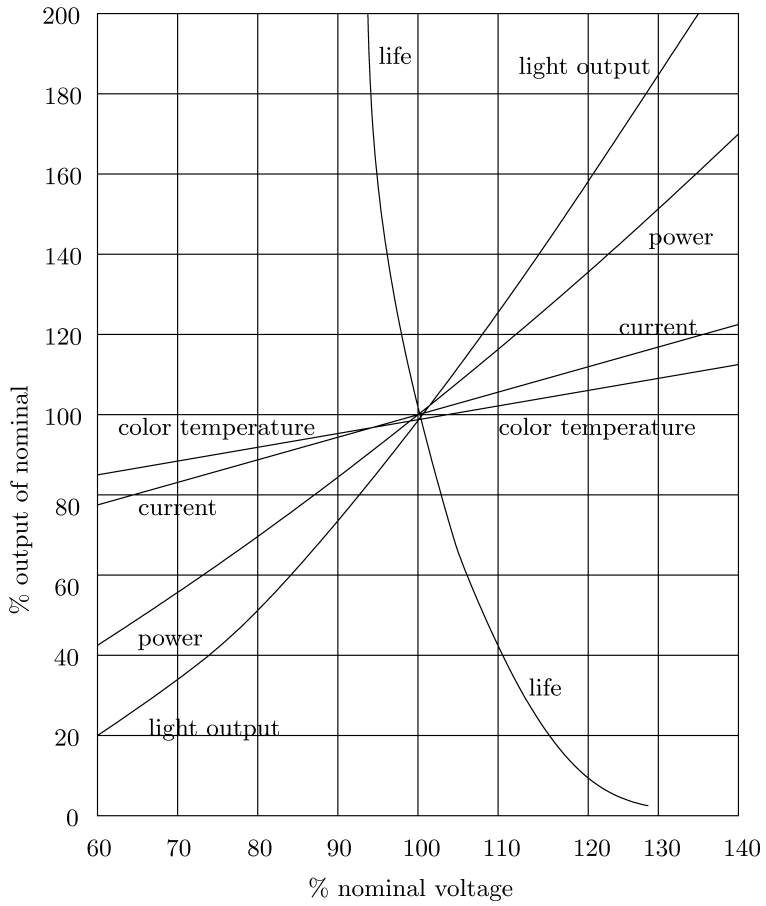

Figure 2.1: Partial characteristics of a tungsten halogen lamp.

lamps have a considerably higher efficiency than tungsten sources, and their life cycle is longer.

However, a discharge source has to be "struck" to start it up by applying a high voltage across the electrodes to break down the resistance within the gas. After that, an electronic ballast regulates the current flowing inside the gas. Striking usually takes about a minute or two; when the lamp is switched off, it needs a cooling-down period before it can be restruck. Some HMI sources with two-sided sockets can restrike immediately, however.

Discharge sources are generally not dimmable. Therefore, fixtures using them use an iris or with a lamelle mechanism for dimming.

Fluorescent tubes are filled with vaporized mercury at low pressure which emits UV light through electron excitation. A coating of sulfides, silicates, or phosphor converts the UV light to the visible spectrum.

Whereas household tubes are not dimmable, professional lighting tubes often are. A dimmable tube has a heating circuit to keep the electrons excited at low voltages. Special controllers are necessary for dimming which traditionally hook up to standard triac dimmers. Recently, control systems which work directly off a digital input signal have emerged.

Of course, more kinds of lamps exist, but their use on stage rare. Of those, sodium

vapor lights do have occasional use because of their intense yellow-orange light.

2.4 Gels

Color is a crucial part of lighting design, and lighting designers use it frequently. In fact, many shows feature few or no fixtures with unmodified, "white" light.

Changing the color of the light of a fixture is only possible by subtracting parts of its output spectrum. There is no way to generate wavelengths not part of the original spectrum of the lamp. *Gels* or *filters* are typically dyed sheets of polyester or polycarbonate which absorb certain wavelengths. Manufacturers of gels usually put out palettes with dozens, if not hundreds of different colors.

A special kind of gel is the *color conversion filter* that changes the distribution of blues and reds to adjust the color temperature of the light. Moreover, *dichroic filters* work by *reflecting* the wavelengths they subtract rather than absorbing them. Dichroic filters are made of glass with a thin layer of suitable chemical inside them.

A *diffusion filter* is a sheet of frosted plastic or glass fiber used to soften the edges of the light beam. Directional diffusion filters stretch the exit angle along one axis.

2.5 Theatrical Fixtures

In theater, the main goal of lighting is the illumination of the actors and the props, conveying the action on stage. Nuances are at a premium, and even small stages often employ large numbers of fixtures to create the lighting for a show. Hence, a theatrical fixture usually fits a quite specific purpose. As moving-light effects are rare in theater (and moving lights are expensive), control over all parameters save for intensity is manual for most of these fixtures. Most stages have electrical installations consisting of a dimmer array somewhere in a back room, with power lines running from the dimmers to a rigging matrix under the ceiling.

2.5.1 Cycloramas, Floodlights, and Groundrows

Floods form the simplest fixture type: a flood consists only of a lamp, a reflector, and housing. The beam produced by a flood is large, as is its exit angle. Hence, floods light large areas on stage, and are not suitable for any kind of selectivity. Nowadays, most floodlights are *linear*, containing a long, horizontal lamp, creating increased spread as compared to a round bulb. Moreover, asymmetrical floodlights feature a larger exit angle at the bottom to ease lighting walls from a fixture hanging from the ceiling.

Figure 2.2 shows a *cyclorama flood* designed especially for lighting the large cloths forming the wall of a stage. The figure also shows a more generic floodlight.

Figure 2.3 shows a special kind of flood, the *groundrow*, which is located on the floor rather than at the ceiling. Groundrows are usually *compartment floods*, consisting of a row of small floodlights.

Parcans Figure 2.4 shows a *parcan*, a simple fixture consisting of a so-called *par lamp* and a simple tin or aluminum housing. A par lamp is an integrated unit with a reflector and a lens sealed in. A par lamp produces a near-parallel beam, with different lamp types producing different beam angles. There is no direct control over focus which is entirely determined by the choice of the lamp. Since the par lamp is a completely integrated unit it can run at high pressure—par lamps are particularly bright and brilliant which makes them uniquely suitable to a number of applications, specifically in concert lighting where bright, parallel beams are important.

Figure 2.2: Cyclorama Flood (Strand Lighting Iris), Floodlight (Strand Lighting Coda).

Figure 2.3: Groundrow (Strand Lighting Orion).

2.5.2 Plano-Convex and Fresnel Spots

The most common type of fixture in theatrical use is the *plano-convex spot*, or *PC* for short. "Spot" is a generic term for all fixtures which allow control over the exit angle of the light beam. Figure 2.5 shows such a PC. A reflector is behind the lamp, and a fixed-position lens is at the front of the housing. The lens is flat on one side and convex on the other, giving the fixture type its name. The distance of the lamp to the reflector and the lamp is adjustable, giving control over exit angle.

The lenses used in PCs often have a pebbled surface which diffuse the beam, giving it a softer edge, while retaining its selectivity. PCs are usually equipped with *barndoors* (as is the one in Figure 2.5)—a set of four rotatable shutters at the front of the fixture which allows some control over beam shape. Designers mostly use barndoors to block out scatter light.

A variation of the PC is the *fresnel spot* which uses a Fresnel lens instead of a plano-convex one. Fresnel spots are shorter and lighter than PCs, but produce a softer, less-defined beam, but are otherwise comparable.

Figure 2.4: A par can (KLS Parcan 64).

Figure 2.5: PC spot (Strand Lighting Alto).

2.5.3 Profile Spots

Figure 2.6: Profile Spot (Strand Lighting Optique).

Profile spots have a reflector-lamp-lens arrangement similar to a PC. However, unlike a PC, a profile spot has a stationary lamp and a movable lens. A profile spot, having a lens with a smooth surface, can produce a narrow beam with a very hard, precise edge. Profile spots have an adjustable aperture just in front of the lamp called the *gate*. A gate can accommodate a number of beam-shaping devices:

- a set of (typically four) shutters to make a three- or four-sided shape
- an *iris*, an adjustable circular diaphragm to alter gate size, or

Figure 2.7: Gobos.

- a *gobo* to create a pattern (see Figure 2.7).

An extension of the basic concept of the profile is the *variable-beam profile* or *zoom profile* which contains two, independently movable lenses in the housing. This arrangement allows independent control over exit angle and focus. Figure 2.6 shows such a zoom profile; the two lateral knobs are attached to the two lenses.

A special kind of profile spot is the *follow spot*, used to create tight illumination of a moving target. A follow spot is mounted on a railing or tripod, and has handles for control by a dedicated operator.

2.5.4 Fluorescent Tubes

Fluorescent tube lamps emit light very different from that of traditional incandescent lamps. Their light is undirectional and very uniform. This makes fluorescent tubes unsuitable for most traditional applications of lighting. However, since they remain cold during operation, they can serve as scenographic elements on stage.

Recently, directional fluorescent fixtures have appeared on the market with similar characteristics as the traditional incandescent fixtures. They have not yet become common on the market, however.

2.6 Color Changers

The next step up in remote control from a dimmable fixture is the *color changer*. Two basic types of color changers exist:

- A rotatable wheel is mounted at the gate which contains a discrete number of gels. The operator can control which of the gels is in the path of the light beam, and therefore choose a color from a fixed selection of about 6–12.
- Several independently rotatable wheels are mounted at the gate, each with continuous coloring. Typically, there are three wheels containing shades of cyan, magenta, and yellow to provide coverage of a large part of the spectrum.

2.7 Beam Shape Control

A number of devices are available to control beam shape:

- A *shutter* can blackout the light very quickly as well as provide strobe-like effects.
- A movable lens can control focus and exit angle.
- A wheel with gobos can provide variable patterns.
- A variable frost can soften the edge of the light beam.

2.8 Moving Lights

Moving-light fixtures provide remote control over the direction of the light beam. There are two basic types of moving-light fixtures, the the *moving head* and the *moving mirror*. With a moving-head fixture, the lamp and the optical arrangement itself moves. The lamp of a moving-mirror fixture remains stationary; its light reflects off a movable mirror.

Moving lights are developing to the point where they are usable even in theatrical environments. One of their main problems is noise generation—moving lights usually need a cooling fan. The noise generated by the stepping motor is also sometimes a problem.

2.8.1 Moving Heads

With moving-head fixtures, the parameters for direction control are *pan* and *tilt* as dictated by the construction of the yoke. For a moving head hanging "upside-down," pan rotates the fixture in the horizontal plane, whereas tilt changes the vertical angle. Figure 2.8 shows such an arrangement.

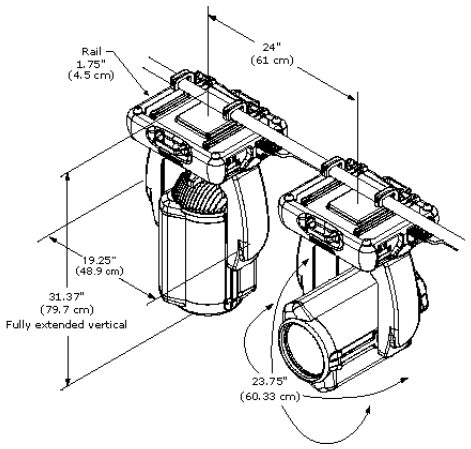

Figure 2.8: Moving head (Vari-Lite VL2400).

Figure 2.9: Moving head for theatrical usage (Strand Lighting Pirquette).

Simple moving heads are basically standard theatrical fixtures equipped with motors for pan/tilt control. Figure 2.9 shows such a fixture; it is obviously a beefed-up PC.

Sophisticated moving lights, mostly for concert usage, include an entire array of controls in addition to direction alone. These fixtures also feature gobo changers, color changers, adjustable focus and beam size, and shutters. Figure 2.10 shows such a fixture.

Note that the electrical connections confine the angular movement of both pan and tilt. Usually, tilt range is just over 180°, pan range just over 360°.

2.8.2 Moving Mirrors

A moving-mirror fixture puts the lamp, its optics and electronics in a non-moving base and projects a light beam at a movable mirror. Moving-mirror fixtures are often called *scanners*. Because a mirror is much lighter than the lamp and the optical machinery, scanners can provide significantly faster moving effects. This makes them primarily useful for disco and concert lighting. Since the light emitted by a scanner is usually quite narrow and focused, it is mainly useful for specialized tasks and for replacing zoom profiles.

Figure 2.11 shows a moving-mirror fixture. It illustrates that the moving mirror

Figure 2.10: Moving head for show usage (Martin MAC600).

Figure 2.11: Scanner/moving mirror (Martin RoboScan 918).

has tighter movement restrictions than the moving head.

2.9 Strobes and Other Effects

Strobes are special fixtures which give out a periodic series of very short light flashes. This creates an impression of jerky motion.

Other light-related effects include:

- slide, overhead, or video projections
- mirror balls
- pyrotechnics
- artificial fog
- "black lights," UV light directed at materials that fluoresce under it
- lasers

Chapter 3

Protocols for Lighting Control

Little Miss Muffet sat on a tuffet,
eating her curds and whey ... Along
came a spider and sat down beside her,
and extended his hand out to play.
—Mr. Reindeer in *Wild at Heart*

Any system for lighting control must interface to the actual fixtures in use or to the electrical installation driving them. The lighting industry has created a multitude of protocols and protocol standards for such interfaces; a fair number of them are still in active use. The characteristics of these protocols determine the structural design of both interface hardware and the software drivers for that hardware. Lula is no exception in this regard—its development produced two separate hardware interface designs, as well as half a dozen or so revisions of the driver structure. Moreover, Lula can now drive a small zoo of commercial hardware interfaces (see Chapter 9 for details). Hence, a discussion of common protocols and their characteristics is in order. This chapter is it. By nature, this chapter is concerned with structure rather than details. The reader interested in the latter is referred to the literature [Sandström, 1997, Huntington, 2000].

3.1 The Control Problem

Chapter 2 has shown that lighting fixtures are complex devices: any ambitious control protocol which claims to support these fixtures must be sufficiently general to support all or at least most of their functionality. As fixture functionality increases, the fixture control problem also grows:

1. The simplest setting is, again, an array of purely theatrical fixtures. The fixtures typically connect to dimmers (see Section 2.2) which take input from the control system. These fixtures have only one remote-controlled parameter—intensity, essentially a number in a fixed range. With standard theatrical fixtures, the viewer can distinguish somewhere between 250 and 1000 shades of intensity. Moreover, the viewer can distinguish somewhere between 15 and 40 changes in intensity per second as separate.

2. Multi-parameter fixtures require control over more parameters. The numerical parameters—pan/tilt for example—frequently require a resolution greater than that of the intensity control. (At a distance of just 10m, a deviation of

1° in pan angle already moves the focus by 17cm.) Such fixtures generally have their own control interfaces.

3. In addition to the parameter control problem—the fixture merely follows a continually present or continuously retransmitted parameters—there is also a *command control problem*: Some fixtures require discrete commands to change to another state in operation. Examples include firing up discharge sources or setting a mode of operation for the fixture. The control system must only transmit such commands once.

As systems and control setups grow in complexity, two other protocol-related issues arise:

Playback control and synchronization Lighting control falls in the general area of show control which also involves, among other things, sound and pyrotechnic effects. Shows involving several such aspects often require synchronized playback, for example between sound effects and light changes.

Feedback In large lighting setups with many fixtures, it is desirable to have centralized feedback about the proper operation of the installation. Control protocols designed for this purpose must transmit back information about hardware problems—blown bulbs, discharge sources gone out, motor problems etc.

Metainformation With the advent of hugely flexible multi-parameter fixtures, it is desirable that the fixtures hooked up to a control system identify themselves to the system rather than requiring the operator to name and assign them.

3.2 Analog Protocols

Analog protocols are the most simpleminded of protocols. They basically apply purely to intensity control: a small, low-current voltage controls the effective output voltage of a dimmer.[1]

- *Fully-mapped* setups have a separate wire for each circuit, running from the control system to the dimmer. Hence, a setup for, say 48 theatrical fixtures will have 48 lines running from the lighting console to the electrical installation.

- *Multiplexed* protocols periodically transmit several analog voltage signals in quick succession, resulting in packets of intensity values. This requires only a single line. The electrical installation must demultiplex the signal onto the dimmers.

  A multiplexed setup is vastly more practical once the number of fixtures exceeds a certain number. However, there are increased demands on line quality. Also, the electronics involved is considerably more complex. Note that the electrical installation must maintain a small memory—typically a capacitor—to preserve a dimmer intensity between packets.

Many dimmers still allow fully-mapped analog control. The industry seems mostly to have settled for a voltage from 0–10V for a single intensity control.

Also, a number of proprietary multiplexed analog protocols exist. Strand's protocols AMX192 (for 192 channels, later adopted as an USITT standard) and D54 (for 384 channels) protocols became especially popular in 1980s.

[1] A receding faction of dimming systems is current- rather than voltage-controlled.

3.3 Digital Channel Control

The natural move from multiplexed *analog* protocols is to multiplexed *digital* protocols. The only difference at the electrical level is that the transmitted packets consist of digital numerical values rather than values implied by voltage levels.

A number of such multiplexed digital protocols exist. They share a number of characteristics:

- A single transmitter sends packets of numerical values to multiple receivers hooked up to the wire. Thus, the topology of multiplexed-protocol control networks is necessarily DAG-shaped. Protocol support hardware includes boosters, splitters (for shipping a single signal to several destinations), and mergers (for combining the packets of several signals into one by some arbitrary strategy).

- The transmitted packets have no intrinsic structure; they are merely sequences of numbers.

- The association between number positions in a packet (so-called *slots*) happens at the receiving end: All receivers receive all the slots, but only pick out a few.

- The control system re-transmits the packets periodically.

Whereas the structure of these protocols still reflects the original, limited purpose of controlling intensities only, the digital nature of these protocols makes it reasonably easy to also (ab)use slot values for controlling other parameters. However, the protocols themselves contain no provisions for structuring the packets according to these requirements.

A number of proprietary such protocols exist, among them examples CMX (Colortran), K96 (Kliegl) and the AVAB protocol. By now however, they have all but been replaced by DMX512. The latter is sufficiently omnipresent in the industry to deserve its own section.

3.4 DMX512

DMX512 [DMX512-1990, 1990, DINDMX, 1997] is a standard developed by the lighting industry starting in 1986, and turned into various official national standards in 1990. Here is how it basically works:

- On the electrical level, DMX512 is a serial protocol based upon EIA-485, the same standard as used for Ethernet, for example. Its transmission speed is 250 KBit/s.

- The control system periodically transmits packets of up to 513 bytes or slots. The first byte is reserved; 512 slots remain for actual control information.

- There is no feedback and no handshake; the receivers are responsible for decoding packets and picking out the slots meant for them.

- The standard requires receivers to retain the values of transmitted slots for at least one second, but not indefinitely.

- The packets can contain less than 512 slots. With full-sized packets, the maximum transmission rate is about 44 packets per second.

The latter attribute of DMX512 requires transmitters to keep transmitting packets, even if their content does not change. Conversely, when a DMX512 signal stops, or if the gap between packets exceeds one second, dimmers and fixtures have the right to stop operation. This unfortunate property, combined with the susceptibility of the electrical side of the protocol (proper bus termination, interference) frequently leads to connectivity problems, inherently unreliable setups and wild-goose-chase debugging. The lack of feedback is especially regrettable in this context. Moreover, since there is no telling whether a given fixture on the bus has received a packet, DMX512 is less than optimal for command control.

A new version of of DMX512 is in the making [DMX512-2000, 2000]. However, compatibility requirements as well as quibbling between the vendors involved in the process has prevented exciting developments. The new standard will, however, have rudimentary provisions for feedback and the transmission of text.

In the special context of interfacing standard computer hardware with a DMX512 bus, special issues arise which reflect in design choices of the interface hardware and corresponding requirements for the driver software:

- *Volatile* devices which rely on software to actually drive DMX512 packet generation. Volatile devices are often straightforward converters from a standard hardware protocol such as Centronics or RS232C to DMX512. The second hardware interface which grew out of the Lula project, the Lula DMX , is an example. They do not have internal buffer memory. Thus, volatile devices will cease to produce output when they do not receive input.

- *Persistent* devices have an internal buffer memory for the packet slot values. The hardware continually generates DMX512 packets from the buffer. The computer controls the DMX512 packets by modifying the buffer memory.

3.5 Feedback Protocols

The next step up from simple multiplexed protocols is the addition of feedback, providing the control system with information about the proper operation of the installation, advising control system and operator about real and potential problems.

Since feedback already implies bidirectionality, it is only a small step to truly bidirectional protocols. Now that networked computers have become so ubiquitous and with the simultaneous success of Ethernet, most current developments of such protocols indeed use Ethernet as a substrate. Examples include AVAB's AVABnet, Strand's ShowNet and ETC's ETCnet.

However, no clear standards have crystallized as of yet. ESTA's control protocols working group is, at the time of writing of this dissertation, working on *ACN—Advanced Control Network*—an ambitious standard for building lighting control systems [ESTA2000, 2000]. The ACN effort includes provisions for the following facilities:

Automatic Resource Allocation Special nodes in the network—so-called *session managers*—manage the allocation of protocol slots to other nodes in the network.

Error Detection and Diagnostics Recipient nodes of control can communicate back error information and diagnostics.

Dynamic Configuration The network can react to and deal with changes in the membership of the network.

Resource Sharing The network can include several control systems.

Metainformation Devices hooked up the network can communicate their functionality and parameter mapping to the control system(s). A special *Device Description Language* (*DDL*) is being invented for this purpose.

3.6 Playback Control and Synchronization

The final issue in protocol design is the coordination between different components of show control. Two separate approaches to the coordination problem exist:

Absolute time One means of coordination is to use absolute, coordinated time. Two standardized protocols exist for this purpose: the ANSI-standardized SMPTE Linear Time Code, and the MIDI Time Code that is part of the MIDI standard in music [MIDI, 1996].

Synchronization The other means of coordination is sending playback signals from one control system to another. The most popular method for doing this is MIDI Show Control (MSC) designed specifically for this purpose.

Chapter 4

Basics of Lighting Design

You see, Johnnie. I toucha number one bottle once, I toucha number two bottle once, and I touch your face. This is a game we love to play.
—JUANA in *Wild at Heart*

This chapter gives a short account of the technical basics of lighting design. The material presented here draws from established sources in the literature on the subject but puts a special emphasis on the specific aspects affecting the design of lighting design systems. It is directed at readers who have had no or little prior experience in lighting design to provide a feel for the process of lighting design.

The presentation exhibits a bias towards issues affecting then design of control system. Consequently, the treatment of some areas have been simplified compared to more comprehensive treatments of the subject. Specifically, the chapter contains no material on matters such as production style analysis, designer-director communication, or safety. The interested reader may refer to the extensive body of published literature [Reid, 1987, Shelley, 1999, Reid, 1993, Gillette, 1989, Fitt and Thornley, 1992].

Furthermore, lighting design is both an art and a craft. At its best, stage lighting possesses expressive power comparable to that of an actor. Consequently, no single, true "method" for creating the lighting for a stage production can exist. The field has many rules of thumb and general guidelines—proficient designers break them all when it fits their purposes.

4.1 Purposes of Stage Lighting

Stage lighting differs radically from conventional room lighting, both in its range of purposes and its complexity. Consider the following selection of functions stage lighting can perform:

Illumination Lighting is necessary to make things and persons visible on stage.

Modelling Lighting serves to highlight three-dimensional structure, of actors' faces, dancers' bodies, set pieces, set spaces, etc.

Focus Lighting can emphasize people, specific areas, and set pieces at the expense of others.

Representation of the Environment Lighting can represent aspects of the environment such geographical location, season, weather, time of day, and temperature through lighting characteristic to these conditions.

Creation of Atmosphere Human emotion reacts strongly to light as a source of mood. Stage lighting can exploit this.

Transition Lighting often changes to mark transitions in the chain of events on stage.

Pacing Animated light can provide pacing to a stage show.

Hiding Lighting can hide things or events happening on stage, for example by drawing away the focus from them or by blinding the audience momentarily.

Painting Especially in concert lighting, the light itself can be the focus of audience attention rather than objects it might illuminate. This applies specifically to animated, moving light.

4.2 Lighting a Subject

For a given show or event, some part of the lighting will usually be chiefly for illumination—making someone or something on stage visible. At first, this may seem a simple job—just point some fixture at the subject head-on. Unfortunately, this is rarely sufficient or appropriate. This section focuses on lighting a single person on stage.

4.2.1 Directional Lighting

For lighting a single subject, it is important to consider the specific effect of light coming from a single direction. Mere visibility is usually enough: for an actor,

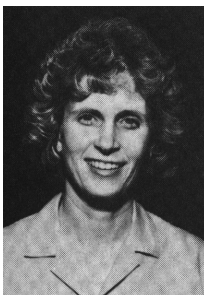

Figure 4.1: Front light [Gillette, 1989].

the audience needs to see facial gesture. Directors in theater often use spatial arrangements to illustrate relationships. For understanding dance, it is crucial to view it in three dimensions rather than just two. Figure 4.1 shows a with head-on so-called *front light*. It appears flat; the light hits the face at right angles, gets into most of the facial crevices, thus effectively obliterating most facial features.[1]

Changing the direction of the light to come more from the side improves the visibility of facial features. Unfortunately, now one side of the face is much more visible than the other—depending on the specific angle of the light. This is usually an unwanted effect, and the resulting image looks unnatural. The specific effect of emphasizing the three-dimensional aspects of a face, body or set piece is called *modelling*.

[1] Drastic make-up can recover some of the facial structure, but at an obvious cost.

As far as illumination and modelling is concerned, using a single light source from *any* direction usually has desirable as well as unwanted effects. Here is a summary of the directional possibilities, as seen in the horizontal plane.

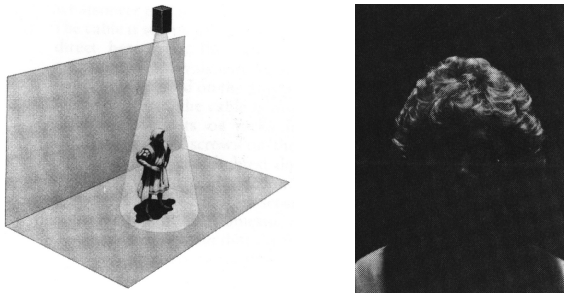

Figure 4.2: Down light [Reid, 1987, Gillette, 1989].

Down light Down light is vertical light from above. It provides a very specific focus on the actor, but does not illuminate his face. As the light moves slightly to the front, facial features become visible. Facial extremities—typically the the eyebrows, the nose, and the chin throw dramatic shadows.

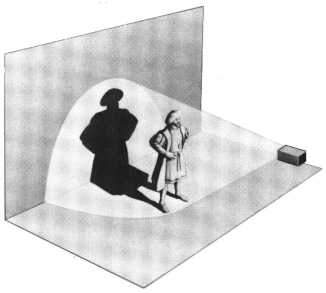

Figure 4.3: Front light [Reid, 1987].

Front light As the light moves from the vertical axis to the front, facial features become more visible. However, as it reaches the horizontal, just like with down light, the quality of the illuminated face will decrease. Moreover, horizontal front light illuminates a corridor behind the actor, throwing a large shadow.

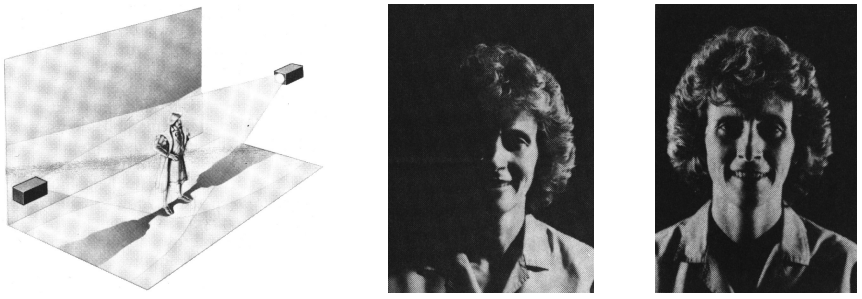

Figure 4.4: Side light [Reid, 1987, Gillette, 1989].

Side light As lighting moves from the vertical to the side rather than the front, both modelling and visibility increase on the side the lighting comes from. As with front lighting, the more the light moves towards the horizontal, the larger the area illuminated will be.

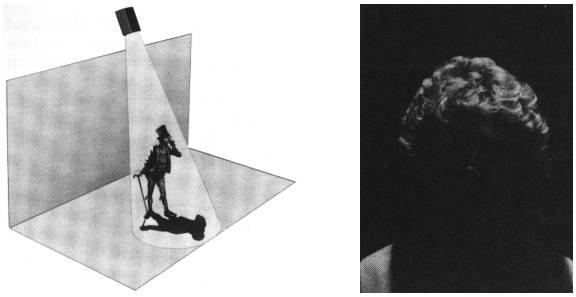

Figure 4.5: Back light [Reid, 1987, Gillette, 1989].

Back light Light coming from behind the actor does not illuminate the face directly, but creates an enhanced impression of depth.

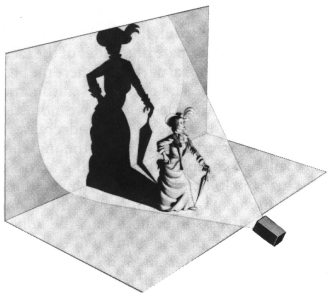

Figure 4.6: Light from below [Reid, 1987].

Light from below In most stage environments, light from below will necessarily have to come partly from the front. While such light generally has a similar effect on face visibility and modelling as down light has, the reversal of direction creates unnatural-looking effects. Hence, lighting designers usually employ light from below to achieve very specific effects. Moreover, lighting from below creates shadows bigger than the subject.

Of course, the actor will not always face the front, and, by moving, change the specific direction of the incident lighting. The general effect of light coming from a single direction is always the same, however.

4.2.2 Composite Lighting

The discussion of the previous section suggests that a single source of light is usually not sufficient for lighting a person on stage. Instead, a lighting designer usually combines several fixtures to achieve composite lighting with good modelling properties, hopefully eliminating adverse effects emanating from specific light sources.

The standard approach to lighting a subject on stage is to use three fixtures, two coming diagonally from the front, one coming from the back, with approximately

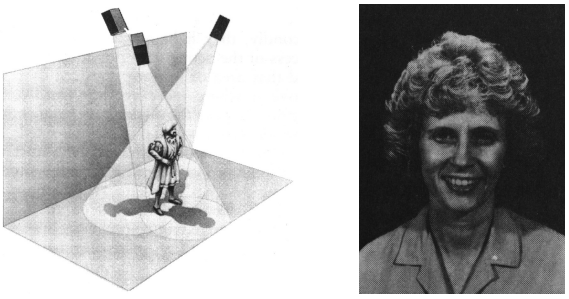

Figure 4.7: Light from three primary angles [Reid, 1987, Gillette, 1989].

120° between them. Figure 4.7 shows this arrangement. Since the fixtures usually have independent dimming controls, the designer can vary modelling and other aspects of the lighting, such as directional motivation simply by changing the relative intensities of the lights involved in lighting a single subject.

Many variations on this combination exist: the exact incident angle of the back light is not important—especially on small stages, it is often a straight down light. Also, the exact angle of the two crossed lights from the front may vary. Often, the stage arrangement will prevent exact adherence to the arrangement—at the sides of the stage it is often not possible to position a third light in an appropriate position.

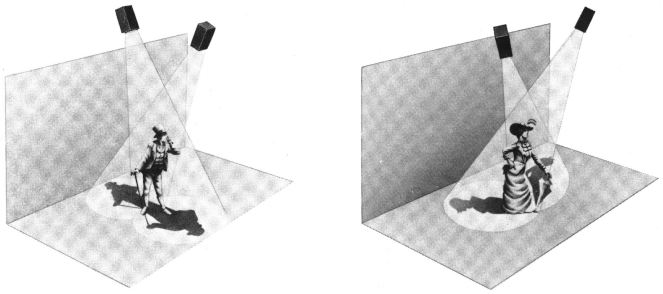

Figure 4.8: Two light sources [Reid, 1987].

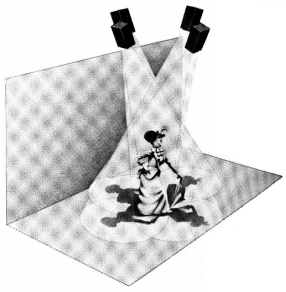

Figure 4.9: Light coming from four primary angles [Reid, 1987].

Furthermore, balanced lighting is also, to some degree, possible with two lights (see Figure 4.8). Four lights as shown in Figure 4.9 provide even more flexibility.

4.3 Color

Color is a crucial ingredient in lighting design: a subtractive gel attached to the front of an fixture changes the color of the light it projects. Color is a powerful means for influencing what the audience sees. Color conveys information about our environment, and influences our mood. Lighting designers employ gels frequently for a number of purposes:

Colored light This is the most obvious use of color: create light of specific, perceptible color.

Canceling out non-linearities The light from incandescent sources most common in stage lighting changes its color as the intensity changes. The brighter the glow, the more its color spectrum moves towards white. At lower intensities, it will move visibly towards yellow and red. Since gels have a subtractive effect on the light passing through them, they can balance the spectrum across the intensities.

Modelling Having several lights trained on a single subject as explained in the previous section means that the designer can equip them with different colors. A careful selection, for example of peach and rose colors with steely blues for the front light, can enhance the modelling qualities of the light without visibly projecting a specific color.

Atmosphere Colored light can create atmosphere and convey a mood. Warm colors in the yellow/red spectrum are "cheerful," colder colors, green and blue, create an unhealthy, sad atmosphere.

Environment Colored light can convey environmental attributes, such as time, place and weather. Even subtle aspects of the environment such as humidity have their expression in lighting.

Note that unchanged "white" lighting from stage lighting fixtures frequently looks unnatural, as it does not correspond exactly to any light either in natural outdoor lighting or typical indoor lighting.

4.4 Secondary Targets for Lighting

So far, the focus has been on lighting a person or stage piece—a (three-dimensional) subject. Lighting makes the subject visible, and helps in defining its visual qualities. Some light will be focused on different targets, however, depending on the function it is to fulfill.

4.4.1 Atmosphere and Environment

Whereas light on the subject focuses on making someone or something on stage visible, atmospheric and environmental lighting works by some quality of the light itself rather than of something that reflects it.

Conveying atmosphere and environment through light is often a function complementary to the primary task of lighting a subject. Hence, lighting designers will often use separate fixtures to create and change atmosphere and environment during the course of a show.

4.4.2 Walls and Backdrops

Walls and backdrops[2] are usually flat surfaces. Backdrops may have elaborate scene paintings on them that require as much attention as human subjects do. With the rich textures common for these surfaces, the direction of the lighting does not much matter. Separate floods positioned above or below the surfaces do the job.

4.4.3 Simulating Real Light Sources

Often, a set implies the presence of an actual light source, such as the sun, the moon or lamps positioned inside a room. Especially in naturalistic sets, special fixtures create effects similar those the actual light sources create. It is rare that an actual light fitting—a *practical*—is sufficient to create its own effect. Typically, the rest of the lighting is relatively much brighter than household lights, and the designer adds further theatrical lights to complete the picture.

4.4.4 Effects and Projections

Special lighting fixtures can create a wide range of special effects. Here are some examples:

Projections A light projects a slide or video onto a solid or semi-transparent screen (from the front or from the back) or onto smoke.

Shadow projections The lighting designer paints a shape on a rigid piece of transparent material and uses a light to project the resulting gobo onto a surface.

Strobes Strobe lights give a fast series of light flashes, making the action appear to be frozen into a jerky series of still frames.

Black lights Black light is suitable for scenes where only very specific objects or part of them are visible.

Gauze Gauze curtains, when lit only from the front appear opaque. Removing the front light and lighting the scenery behind the curtain can make it "magically" appear.

4.4.5 Corrective Light

Some lights have the sole function of eliminating odd effects created by the rest of the lighting installation. They serve to eliminate unwanted shadows and blind spots.

4.5 Lighting the Stage

The previous sections have primarily dealt with lighting a single subject at a time, or creating a single effect at a time. Ultimately, the designer needs to light the entire stage, resulting in a complete stage picture—a so-called *look*. This section describes a common procedure for placing and combining the available lights to create a coherent look. Again, hard rules do not exist in the field. A designer might employ an entirely different approach.

[2] A *backdrop* is a large cloth at the back or the side of the stage, sometimes with a painted motif.

4.5.1 Playing Areas

Even though a finished look might create the impression of uniform lighting when uninhabited, a given production will have certain focal points: places where actors or musicians perform crucial actions, or stay for longer amounts of time. The stage action might dictate the choice of focal spots. Also, seating furniture, doors, or bars might create natural areas of focus. Typically, a designer will determine the playing areas from the groundplan and from discussions with the director or stage manager.

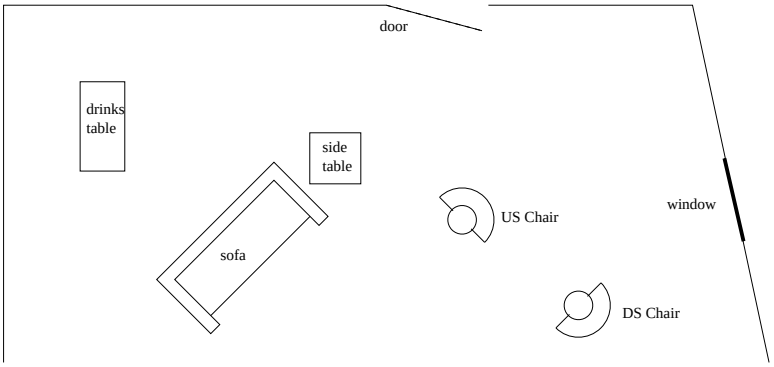

Figure 4.10: Sample groundplan.

Figure 4.10 shows a sample groundplan. The furniture—the sofa and the two chairs ("DS" is for "downstage" meaning closer to the audience; "US" correspondingly is for "upstage") are the focal points of three natural playing areas, as are the drinks table, the side table, the door, and the window.

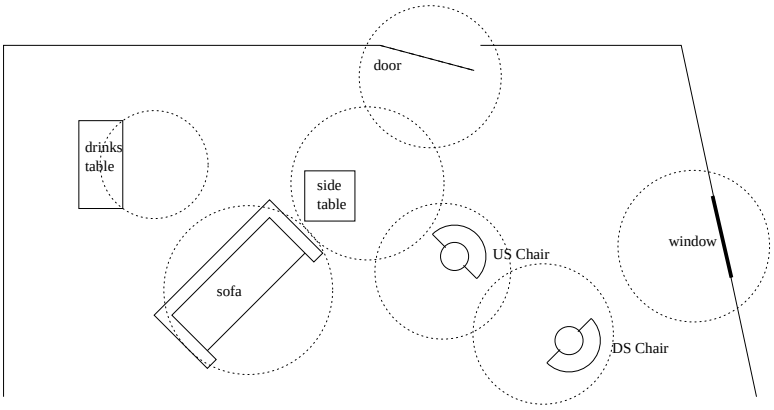

Figure 4.11: Sample groundplan with playing areas.

Lighting the focal spots creates *playing areas*—usually roughly circle-shaped areas on stage. The size of the playing areas depends on a number of environmental factors: the distance between the fixtures and the area, the opening angle of the fixtures used, use of shutters etc. A diameter of 6–10 feet is normal for a single playing areas. Figure 4.11 shows the sample groundplan with the central playing areas drawn in.

Often, the playing areas intersect or create gaps between them. Intersection requires careful coordination in colors and intensities of the intersecting areas. How-

ever, at this stage of the design process, it is usually not a matter of concern yet. In some cases, floor coloring or texture will have a strong effect on the visual impressions created by intersections. In that case, some prior planning may be in order.

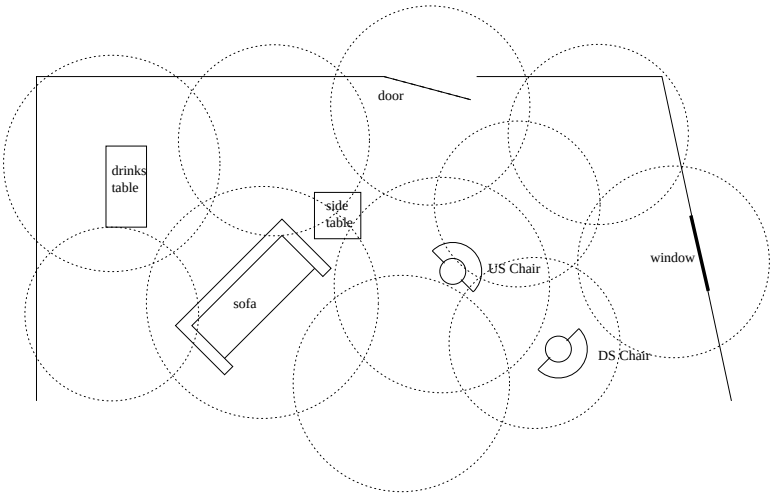

Figure 4.12: Sample groundplan with primary and secondary playing areas.

As for gaps, they typically create require separate lighting, creating secondary playing areas. Extending and slightly moving the primary playing areas where appropriate, and adding secondary areas creates complete coverage of the stage. Figure 4.12 shows the modified groundplan. Note that the primary playing area around the drinks table has disappeared—there is no way to place the secondary area without lighting the table separately. As long as no pointed focus on the table is necessary, this arrangement should be sufficient. A similar rearrangement is possible for the side table. The small gaps still remaining will either disappear "by themselves" by adjustment of the lights for the existing areas; where necessary, the designer can add additional fixtures to fill the gaps later (see Section 4.5.3).

The division of the groundplan into playing areas is the first step in the design process, creating a basic lighting arrangement. The planning of atmospheric and environmental lighting, practicals, and special effects is separate from this. In the particular arrangement of Figure 4.12, it might be suitable to plan for additional back light beyond the door and the windows.

4.5.2 Fixture Placement

Next on the list is determining positions for the fixtures. This happens with the help of a special groundplan which includes the pipes and booms and all other rigging equipment, a so-called *rigging plan*. Occasionally, it is not possibly or inordinately costly to move the fixtures. In that case, determining fixture placement reduces to assigning the existing fixtures to functions within the lighting plan.

The plan in Figure 4.12 shows that even a small production, involving only handful of playing areas and specials, requires a large number of fixtures, especially when the designer strives for three-source or even four-source lighting for any given area.

Figure 4.13 shows a typical arrangement from such a plan. Three adjacent playing areas require two front lights each, coming in at an angle. This results in a

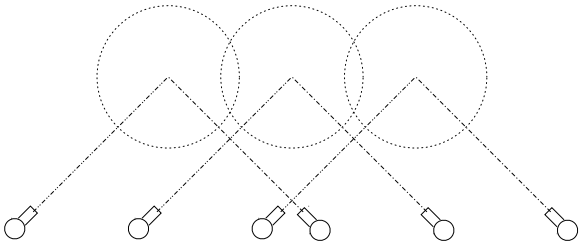

Figure 4.13: Fan setting.

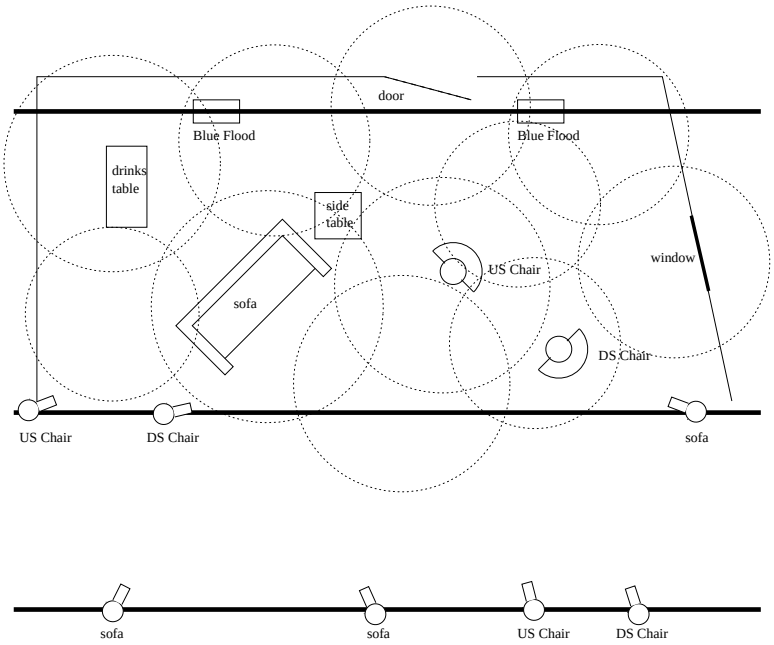

Figure 4.14: Partial rigging plan.

so-called *fan setting*, where fixtures lighting different playing areas are interleaved on the pipes.

Complete rigging plans are usually large. The dense arrangements might even involve several overlays. Figure 4.14 shows the beginning of such a plan, with three-angle lighting for the sofa, two-angle lighting for the chairs, and two blue floods upstage.

A complete rigging plan will also indicate gel colors, and the assignment of dimmer circuits to fixtures. In environments with a limited number of circuits (this actually being the norm), it might be necessary to switch several fixtures to a single circuit. In this case, the rigging plan also includes the relevant information.

Once the rigging plan is finished, the actual onstage work can commence: the fixtures are hung and connected.

4.5.3 Coloring and Focusing

The next phase in lighting is pointing each fixture at its target, making the necessary adjustments to the shape of its beam and inserting the appropriate color gel. This process is called *focusing*.

Since with most fixtures the main concern is lighting actors, the lighting designer will walk around on stage during the focusing. (When the set is particularly tricky or spectacular, it might be necessary to light the set per se first and determine later if the resulting lighting is sufficient for the actors.) An operator will turn on the lights one by one, and the designer will usually use her shadow (or a pair of sunglasses) to check that the light is actually hitting her.

The secondary concern with focusing is to curb unwanted light: almost every fixture will throw light at places which do not need it, or, worse, where it creates an unwanted effect. When the designer is lucky, the stray light is mostly on the floor or offstage where it does not cause any harm (unless the floor is white ...). When it is not possible to completely block out the unwanted light, the lighting designer typically applies two principles to solve the problem:

- Soft edges are less noticeable than hard ones. PCs and Fresnel spots allow the necessary adjustments to create a soft edge. For profile spots, frosts and diffusion filters do the job.

- Where possible, beam edges should coincide with edges on the set, or on the proscenium. Almost all theatrical lights have shutters or barndoors for this purpose (see Chapter 2).

For a lighting designer, scatter light is the enemy as it is not under the control of the designer. Because of this, most stage floors and bare stage walls are black.

After the individual lighting of each playing area, the designer still needs to check if there any gaps (sometimes called *black holes*) remain between them. Eliminating them might require readjustment, and, in some cases, hanging additional fixtures.

The rigging and focusing phases are often interleaved.

4.5.4 Intensity Adjustments

With multi-angle light for a single acting area, the fixtures involved will need separate intensity adjustments to achieve smooth illumination as well as good modelling. Therefore, making and noting down the relative intensities needed for lighting each playing area is next on the plan.

The lighting for the separate playing areas, merely taken together, will typically not result in a smooth look for the entire stage. Even though each area has a good intensity balance "internally," they will not convey identical intensities when seen together.[3] This requires another round of adjustments, keeping "internal" relative intensities constant and only varying "external" intensities. Moreover, the coloring or beam shape of adjacent playing areas might not work well together, requiring more fixture adjustment.

When the basic lighting is finished, it may be necessary to introduce specials to eliminate unwanted shadows. Finally, the designer will put the basic light in conjunction with the atmospheric and environmental lights, and add the other effects to create an arsenal of looks. Ideally, the arsenal will cover the entire range of lighting situations during the show.

4.6 Assembling the Show

The final phase of preparation for a show is arranging the finished looks into a sequence. This brings time into the purview of the lighting design. Most shows are essentially sequences of scenes with fixed looks and with controlled transitions

[3] "Perfect sight" seems to be almost as rare as perfect pitch.

between them. Such a transition is called a *fade*. As looks by themselves convey static properties of the stage picture, transitions between them show change:

- Simple transitions simply separate scenes from each other. (Actually, the simplest of all transitions are blackouts for the sole purpose of allowing set changes to happen in the dark.)
- A change in focus can draw the attention of the audience from one aspect of the stage to another. For a set with several locations present on stage at the same time, a focus change can prepare a shift of the action from one setting to another.
- Subtle changes in atmospheric light can underline atmospheric changes in the action.
- Changes in the environmental light suggests environmental dynamics—the passing of time, or a change in weather.
- Generally, all changes in color can effect transitions in all aspects on which color has an impact (see Section 4.3).

These are only examples—as with the static aspects of lighting, the functions transitions can take on are limited only by the designer's imagination.

The arrangement of fades usually happens in close coordination with some sort of script—this might be a play script or a rough list of what will happen during a show. Cross-references are necessary to ensure that both remain in synch.

In addition to the pre-scripted sequence of transitions, it might also be necessary to arrange for any number of manual controls for aspects of the lighting the show operators needs to operate live. For theatrical productions, the only manual control might be for the curtain call lighting. Concert lighting typically requires large numbers of manual controls.

4.7 Concerts and Moving Light

Concert lighting emerged fairly late into a discipline by itself—in the 1960s. Hence, the specific body of established requirements and techniques for concert lighting is by far not as developed as for theater lighting. Documentation is scarce. Hence, this section can only give the briefest glimpse of concert lighting as it touches upon console design. Again, the reader is referred to the literature for a better insight [Moody, 1998].

Lighting for concerts has some of the same functions as theater lighting: the audience wants to see the musicians on stage. Also, more and more big rock bands erect theatrical sets on stage. However, concert lighting has two major elements theatrical lighting usually does not have: effect lighting and dynamic manual control.

Effect Lighting The main departure in concert lighting is the use of lights not meant to illuminate something on stage. For a concert, most of the lights typically point into the air. The audience sees the light beams themselves, and the dynamic patterns rhythmic patterns they paint as they go on and off and as they move.

This is mostly the reason for the development of the large range of moving lights (see Chapter 2): the beams of moving lights can create startling visual impressions: fans that open and close, sweeps of the audience, circular motions, combined intensity and direction changes like ballyhoos etc. Movement of the lights is, just like a form of dancing, a visual support of the music.

Dynamic Manual Control Whereas departures from a preestablished sequence of events are rare (and usually unwanted) in theater, they are the norm in concert lighting: the band might simply change the order of the songs, improvise or do other unpredictable things.

Consequently, the job of the playback operator is significantly harder than in theater: she needs direct simultaneous manual access to a large number of possible light changes and effects. Several effects might have to happen simultaneously. Moreover, the operator requires additional control over parameters such as the speed of dynamic effects, relative timing, and relative positioning.

4.8 Lighting for Other Occasions

A number of other event types require professional lighting and professional lighting design. Their requirements vary. Some examples:

Dance already mentioned briefly in Section 4.2.1, has all the requirements of theatrical lighting in three dimensions. In addition, dance often requires follow spots or moving lights to create a special focus on a single dancers.

Musicals combine elements from theater, dance, and show lighting. The premium is usually on effects, not nuance.

Variety shows such as galas and award presentations draw from a large variety of presentation forms and, therefore, elements from all disciplines of lighting.

Industrial presentations are usually instances of show lighting: effects are required. Special circumstances arise from the fact that the object of lighting is often a thing rather than a person.

Sports events With big sports events, the chief difference is in dimension which requires coordination between many lighting designers and operators.

Chapter 5

Lighting Consoles

That's a double-barreled, sawed-off,
Ithaca shotgun with a carved pistol
grip stock wrapped with adhesive tape.
Next to it's a cold Smith and Wesson
.32 handgun with a six inch barrel.
—Bobby Peru in *Wild at Heart*

The final part of a lighting installation is its control system, or *console* for short. A console is essentially an interface between the (human) operator and the fixture electronics.

The market is abundant with lighting consoles. This chapter gives a systematic account of the functionality requirements for lighting consoles as well as of common industry practices in console design. Moreover, it defines basic, consistent terminology for the basic concepts underlying console design. The terminology set forth here does not correspond to a single accepted standard, as there is none: The manuals for the various lighting consoles as well as books on the subject use different terms for identical concepts. Worse, they assign different meanings to the same words.

The design of a new lighting console such as Lula requires a careful systematic study of existing systems. Hence, the second part of this chapter is an attempt at classifying the properties of consoles available today. The chapter concludes with a survey of some of these consoles.

5.1 Parameter Control

Technically, a lighting console determines all controllable parameters of the fixtures on stage, as described in Chapter 2. At the very least, for purely theatrical consoles, this means controlling the dimmers to affect intensity. Sophisticated consoles additionally know about other fixture parameters such as pan/tilt, color, focus, beam shape, gobos, shutters,etc. Moreover, such consoles often allow the operator to configure further parameters and define how new fixture types allow control over them.

5.1.1 Fixture Types

As described in Chapter 3, existing digital protocols for fixture control do not carry metainformation: the control information is basically a stream of packets, each packet an unstructured vector of bytes. Moreover, they are usually unidirectional—the console transmits, the dimmer or fixture receives and executes.

Consequently, the operator must manually specify the nature of the lighting installation hooked up to the console—how many fixtures it includes, and how it turns a byte packet into a specific setting for its parameters. As just about every fixture type implements its own mapping, the console needs to maintain a library of fixture types with the necessary information to implement the necessary pre-protocol reverse mapping. Typically, the operator, upon starting out on a new venue, tells the console the number of fixtures and their types.

5.1.2 Patching

Traditionally, *patching* is the assignment of circuits to wall sockets—an electrician would connect a circuit outlet to a socket feed via a cable. A similar process is necessary with modern digital protocols for lighting control: a fixture (or a dimmer) will receive its parameter settings from a certain window of the byte packets it receives. As the protocols are unidirectional and do not perform any automatic negotiation of window addresses, their assignment is another manual job for operator: after setting the window addresses on the hardware, she needs to communicate these same settings to the console. A console maintains its own internal addressing scheme—based on numbering or naming—for the fixtures, and so this process consists of assign hardware addresses to console-internal, "logical" addressing. This process is also called patching or *softpatching*. Some consoles allow arrangements other than the usual one-to-one mapping, such as mapping several fixtures to a single internal address.

5.1.3 Dimmer Curves

The conversion of digital value to a light intensity—a physical entity—is a very complex process [Gerthsen *et al.*, 1989]. It is not even at all clear which photometric unit of measurement should actually be the basis for the conversion, whether it be an objective one, or whether the conversion should take into account the spectral sensitivity curve of the human eye.

Moreover, in real lighting setups, the conversion of control into intensity is usually a two-stage process: a dimmer converts some digital quantity into an electrical waveform (see Section 2.2); the lamp or tube converts the waveform into light. Tolerances in the manufacturing processes of both dimmers and lamps are often very visible. Therefore, when the operator specifies an intensity of, say, 50%, this "half intensity" might not translate to any intuitive counterpart on stage. Worse, one fixture "at 50%" can produce an entirely different subjective intensity from another fixture hanging next to it at the same percentage, especially when distinct fixture or dimmer types are involved.

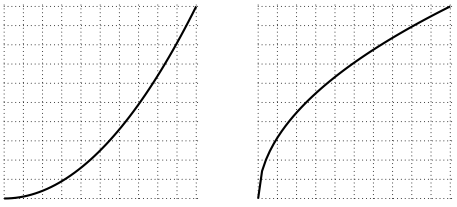

Figure 5.1: Intensity output of a tungsten lamp and dimmer curve correcting for the non-linearity.

Thus, a lighting console needs to be able to correct for this type of *static non-linearity*. Typically, the operator can assign a *dimmer curve* to the intensity pa-

rameter of a fixture, specifying a mapping from the console's unit of intensity measurement to a parameter setting for the fixture. This mapping is effectively an inverse to the mapping performed by the dimmer and the lamp. Figure 5.1 shows the intensity conversion of a typical tungsten lamp along with a dimmer curve that corrects it.

Adjustable dimmer curves can also solve another pragmatic problem: lamp types which require preheating get dimmer curves which start at the preheat intensity rather than at zero.

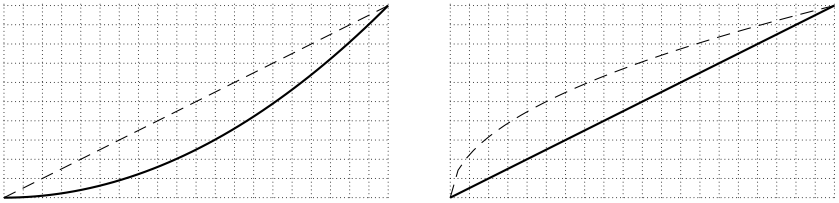

Figure 5.2: Dynamic non-linearity.

Such a simple dimmer curve assumes that the lamp is always in a stable state—its intensity parameter setting as well as the intensity itself do not change. However, transitions between such stable states often happen smoothly over a period of time, sometimes several seconds. Different types of lamps or tubes, in addition to these static non-linearities, also exhibit markedly different dynamic behaviors. Fluorescent tubes react almost instantaneously, whereas big floods usually require large amounts of energy to heat up or cool down, and therefore are usually late in executing requested intensity changes, resulting in *dynamic non-linearities*. Figure 5.2 shows an example. To some degree, it is possible to compensate for these by using a non-linear output ramp. Note that using an inverse to the light output at linear input as suggested by Figure 5.2 will probably not be entirely correct, but might at least help.

5.1.4 Indirect Parameters

The relationship between a number, say, between 0 and 100, and a visible lighting intensity is fairly intuitive. With other parameters, specifying its numerical value directly may not be appropriate. Instead, a console might allow control over a different representation of a parameter—a so-called *indirect parameter*. Examples are Cartesian coordinate control over pan/tilt, or the application of different color models to color changers.

Pan/Tilt Pan/tilt is not a good quantity to work with if the operator wants to hit a specific spot on stage, even more so when the intended target is moving: Cartesian coordinates are more appropriate. Consoles offering Cartesian coordinate control require calibration before programming begins: the operator indicates the pan/tilt settings corresponding to certain reference spots on stage prior to programming.

Color Since color gels work in a subtractive manner, color changers capable of mixing colors usually accept the color parameter as a CMY triple. However, for a human, a different color model such as HSV or RGB might be more intuitive. Moreover, lighting designers and operators are frequently used to the numeric coding of color gels defined by their manufacturers [LEE Filters, 2000, Rosco Laboratories, 2000]. For fixed-size color changers, the operator decides what color gels they carry—this also requires calibration on the console.

5.2 Look

With control over the fixture parameters, it becomes possible to construct complete looks. As looks eventually appear during a show, it is important that the console be able to store looks for later recall. With storage capability comes the need to modify already-stored looks.

5.2.1 Look Construction

Look construction is easily the most time-consuming part of both lighting design and implementation. The operator could theoretically construct a look by setting every single fixture parameter separately. However, this is already tedious with small installations, and unacceptable for large ones—programming the looks would take far too long. Hence, electronic consoles usually have facilities for manipulating identical parameters of several fixtures at once, as well as pre-storing groups of fixtures and parameters for rapid composition. Section 5.7.2 further down describes the various levels of support for look construction in commercially available consoles.

5.2.2 Look Storage

In the old days, the operator would store looks, after having constructed them, by writing down every single parameter setting on paper. Recalling the look would consist of re-setting the parameters in the same way. Of course, modern electronic consoles offer storage of looks for later recall.

5.2.3 Look Modification

Once the operator has constructed and stored a look, it often becomes necessary to change the look later, and replace the storage slot by the new version. Incidentally, many consoles are very optimistic about the need for later changes and make them exceedingly and surprisingly difficult.

5.3 Sequence Assembly

Armed with a collection of looks, the lighting designer can proceed to assemble them into sequence, resulting in the lighting for the complete show. Changes from one look to another are rarely instantaneous; they usually happen gradually. There are several systematic ways of transforming one look into another gradually:

Crossfades A *crossfade* is a smooth linear transition from one look to the next: each intensity will change from the intensity of the original look to the intensity of the new look proportionally to time.

In/out fades An in/out fade has separate times for obliterating and old look and making a new one appear.

In/out fades behave differently for fixtures with increasing and decreasing intensity. For an in/out fade whose in time is shorter than its out time, fixtures which increase in intensity fade more quickly, reaching their target at the end of the in time and staying constant for the rest of the fade. Fixtures decreasing in intensity fade over the longer out time. Figure 5.3 shows an example of such a fade. The (intended) visual impression is that the “old” look lingers around for slightly longer.

Conversely, in/out fades with an in time longer than the out time typically reach a lower overall intensity level before making the new look entirely visible.

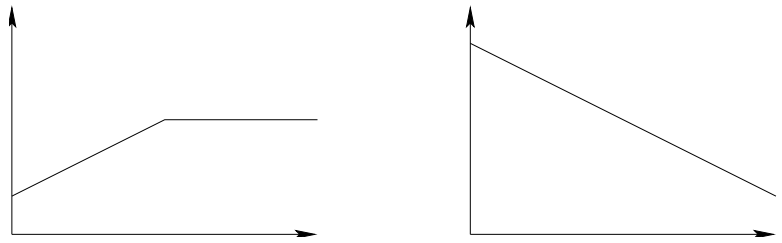

Figure 5.3: In/out fade for fixtures with increasing and decreasing intensity.

The fades of the fixtures with increasing and decreasing intensity behave in an opposite fashion.

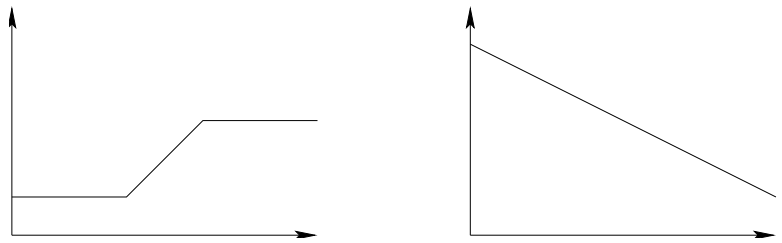

Figure 5.4: In/out fade with in delay for fixtures with increasing and decreasing intensity.

In/out fades with delays Finally, delays allow delaying either the "in" or the "out" part of a fade, actually leaving an old look untouched for the first segment of a fade. As with in/out fades, an in delay affects only fixtures which increase in intensity from the old to the new look. Figure 5.4 shows the fade curves for a delay fade with a short in delay, a short in time, and a longer out time.

5.4 Animated Light

A fade is the simplest and the most common form of *animated light*—light changing automatically over time upon the press of a single button.

Many other forms of animated light are feasible. Fades per se affect only intensity. In addition, animation applies to all other fixture parameters. The common forms of animated light are chases, effects, and shapes, in increasing order of sophistication.

5.4.1 Chases

A *chase* is a sequence of looks, separated by identical fades between them. A chase loops indefinitely, and a variety of ordering choices are common: sequential, backwards, back-and-forth, random, etc. Chases typically implement "flickering" effects such as running lights, burning fires, and disco beats.

5.4.2 Effects

An *effect* a sequence of separate fades, often called *steps*, each to be specified by the operator. Looping is typically optional, but available.

5.4.3 Shapes

Moving lights in principle support *light choreography*—the "drawing" of shapes either with the light beam itself or with the parts of the stage it hits. The first requirement of light choreography is *continuous animation*: the setting of parameters according to continuous functions of time. This applies to movement, but also to color, beam shape, and all other lighting parameters, resulting in *shapes* of movement.

Sophisticated consoles offer the application of a limited range of mathematical function, as well as the construction of freehand shapes. The combination of shapes with effects allow the construction of successions of shapes.

5.5 Playback and Manual Intervention

The crucial job for the console is during the show itself: the console must play back the contents of its memory, performing fades and effects automatically and in real time.

5.5.1 Sequential Playback

For theater productions, playback usually only involves playing back a single sequence of fades; the operator simply presses a "Go" button to initiate each fade, coordinating between the script, the action on stage, and the programming of the console.

5.5.2 Parallel Playback

Big live shows as well as the occasional theater production requires several lighting actions to take place simultaneously. In theater, this is usually necessary when certain parts of the stage obey separate laws as far as lighting as concerned. Subtle, slow, and pervasive changes might indicate the passing of a time or a change in atmosphere. Some illustrative examples:

- A fireplace requires a continual lighting effect which stays the same as the rest of the lighting changes.
- The sun rises during the entire playing time of the show.
- Some part of the stage slowly reveals someone hiding there at the beginning of the show.

5.5.3 External Control

A technically complicated show requires coordination between its various aspects. This is typically a temporal coordination, for example, between sound and light cues via MIDI control sequences, or between light cue playback through a time code. Also, events on stage, such as the tripping of a light barrier might trigger actions on the console.

5.5.4 Look Preparation

In shows where the chief function of lighting is illumination—making visible what is on stage—a fade usually affects intensity only. When moving lights are present, it would be disconcerting if they changed position, color etc. simultaneously. Hence, the operator must ensure that the non-intensity attributes of the target look of a fade are in place before the fade itself happens. Of course, this is only possible with fixtures at zero intensity. This is also job a console can do automatically, at least in principle.

5.5.5 Manual Intervention

Concerts and other "volatile" live shows require that the operator makes lighting decisions during the show itself. In the extreme case, a collection of looks, fades and effects is available before the show, but the operator decides which ones to use at what times dynamically. This requires flexible access to the console's playback facilities, as well as complete manual control over all fixture and effect parameters.

Even theater productions require facilities for manual intervention: actors are occasionally late, mix up the scene sequence, or move to an unexpected place on stage. Also, applause lighting must adapt to what happens on stage and the audience. Here are some common forms of manual intervention:

- temporarily stopping a fade for later continuation,
- changing the speed of a fade,
- changing the rate of change in a chase or effect,
- changing the order of a preprogrammed sequence of fades,
- operating a fader to control lighting of the stage or a part of it,
- "injecting" arbitrary fades or effects into the output of the console,
- turning off single fixtures because of hardware problems.

5.6 User Interface Controls

Consoles offer formidable arrays of user controls for easy fixture and parameter access. Buttons can control boolean values. Continuous parameters, however, require special controls. Here is an overview:

Faders The fader is the simples continuous control. It is *absolute*: its position represents a fixed parameter value. Thus, it also doubles as a parameter view when in use. Moreover, it has fixed minimum and maximum positions.

Rotary encoders A rotary encoder is a wheel the user can turn. Unlike a fader, a rotary encoder turns indefinitely; it has no minimum, or maximum, and is therefore usually a *relative control*. The user cannot see the value represented by the encoder by looking at it; it does not offer a view.

Trackballs, touchpads and mice These UI controls are relative, like rotary encoders, and do not offer a view. They can, however, control two-dimensional parameters. They are usually less precise than rotary encoders.

Motorfaders Motorfaders look just like regular faders. However, the console software can also set its position via a motor.

Figure 5.5: A picture of the GrandMA console [MAGrandMA, 2000].

Touchscreens Touchscreens usually display buttons and allow the user to push them. Some specially manufactured touch screens work like motor faders, however: the user can move her finger along a touch-sensitive display strip. The strip displays the current parameter value.

Figure 5.5 shows a console featuring an especially large selection of different controls.

Many consoles also offer handheld remote-control devices which allow a limited control from onstage, for example.

5.7 Console Functionality Options

How do existing consoles support the requirements of lighting design and playback? Despite the plethora of existing consoles and the truly babylonic diversity in terminology between them, all consoles operate essentially on the same conceptual basis.

They differ in user interface details and quantitative issues—what functionality a console offers, how many fixtures it can control, how many fades it can perform at the same time, memory capacity etc. This section offers an overview of the functionality options that distinguish existing consoles from one another.

5.7.1 Parameter Variety

Consoles differ in their support for the control of different fixture parameters:

Intensity only Traditional theater consoles, particularly those with direct or multiplexed analog outputs are limited by design to control intensity only.

Direct parameter access Consoles with digital protocol outputs such as DMX512 (see Chapter 3) often allow direct control over the values of the output slots. This allows the operator, with the help of the DMX slot assignment chars of multi-parameter fixtures, to control non-intensity aspects of the fixtures. This is, of course, a tedious process. Some consoles contain rudimentary support for discrete parameters, such as color or gobo changers via "non-dim" channels.

Parameter modelling Sophisticated consoles have a conceptual notion of the fixture parameters along with dedicated controls for setting them. Moreover, these consoles maintain a library a fixture types along with their mapping between packet slots and parameter settings.

5.7.2 Look Construction and Modification

The crucial part of a console is its arsenal of facilities for constructing looks. The key to effective look construction is the ability to group collections of fixtures and parameters settings and treat them as entities. Consoles, as they get more sophisticated, offer successively more and more grouping facilities. Moreover, consoles allowing the storage of looks must also allow the operator to modify these looks after construction.

Single-fixture control A pure fader board has one fader per fixture which controls its intensity. Even simple electronic consoles allow only setting the intensity of a single fixture. An incremental improvement is to set several fixtures to the same intensity in one action.

Groups In light painting, fixtures show up in herds: all fixtures in a single row, only the even-numbered ones, all fixtures of a certain type, and so on. During the show, fixtures in such a group often operate in unison: they go up at the same time, or perform the same motions.

In lighting installations with large numbers of fixtures, groups facilitate fixture selection: an operator can set all fixtures of a group to a common intensity, color, position, or other parameter setting. Groups can also help the operator select single fixtures by group.

Submasters A *submaster* is a control for a set of fixtures along with associated intensities. On the console, a submaster is usually associated with a fader which controls the relative intensities of its members. Consider a submaster for a fixture 1 at 30%, a fixture 3 at 70%, and a fixture 7 at 100%, notation $\{1^{30}, 3^{70}, 7^{100}\}$. When the submaster fader is at, say, 50%, console output is $\{1^{15}, 3^{35}, 7^{50}\}$.

Submasters mostly occur in theater: a submaster typically stands for a conceptual part of the lighting, for example the lighting for a single acting area on stage, a certain prop or atmosphere. The operator will typically collect a convenient set of submasters prior to programming proper and use the submasters to construct the looks themselves.

Note that even though an operator might construct a look entirely from submasters, the console will only store fixture/intensity pairs: the storage model of the console is still *flat*. This has some notable consequences:

- When the operator needs to edit a stored look later, the console cannot physically restore the submaster faders to their original positions. Hence, editing again happens at the level of single fixtures.

- Should the operator change the meaning of a submaster, the looks constructed using from it will not change automatically. The operator needs to change each look separately.

Palettes A *palette* is a pre-stored collection of parameter settings. Examples for palettes include:

- a color, representing a specific CMY setting,
- a beam shape,
- a position on stage.

The operator can *apply* a palette to a set of fixtures, turning them all the same color, or focusing them on the same spot on stage.

The stored representation of a look constructed a palette does not contain its parameter settings themselves, but rather a reference to the palette. Consequently, when the operator changes a palette, all looks constructed from it change with it. This is convenient for adapting a show to changes on stage, say, when a position moves and requires refocusing.

Presets A *preset* is a parameter setting for a set of fixtures. As opposed to a submaster, a preset can specify arbitrary attributes, but does not allow relative intensity control. When constructing a look, an operator can *apply* a preset to a set of fixtures; the look assumes the parameter settings specified in the preset.

When storing a look containing a preset, the console stores a reference to the preset itself rather than its individual parameter settings. Consequently, just as with parameters, changes to presets propagate to the stored looks containing them. Still, with most consoles, the presets themselves are still flat.

For some consoles, palettes and presets are the same thing: palettes are presets specifying the settings for a single fixture.

Hierarchical Presets The most sophisticated consoles allow the construction of presets from other presets, creating a hierarchy of presets.

5.7.3 Tracking Consoles

A *tracking console* uses "relative" storage for looks: such a console keeps track of the *changes* an operator makes changing one look into the next. This assumes that look playback will be in the same order as look programming. Tracking addresses two pragmatic problems:

- Tracking effectively implements the sharing of some parameter settings among consecutive cues. This allows the operator to achieve certain changes which must affect a sequence of looks equally by changing the single parameter setting at the beginning of the sequence.
- Playback happens in a "relative" fashion. This makes it easy to prepare looks several sequence positions in advance by setting the parameters of fixtures currently at zero intensity. Tracking will ensure that looks between the preparation and the target look will not change these parameters if they do not set them explicitly.

5.7.4 Effect Construction

Consoles offer four primary methods for constructing effects:

from sequences Sequence effects are essentially sequences of fades. Typically, the same customizations that apply to chases also apply to sequence effects. Most importantly, consoles offer control over the order of sequencing: forward, backward, back-and-forth, random etc.

from functions Another kind of effect arises from assigning mathematical functions to parameters of a set of fixtures. Operators can use this feature to create shape-like movement effects such as circles, ellipses, Lissajous figures, etc. More possibilities arise from the combination of shapes with animated intensity and color. The consoles offering function-generated shapes all provide about the same selection of available functions: standard arithmetic, exponentiation, trigonometry, rectangles, and sawtooth.

Some consoles allow the construction of *composite functions* from arithmetic expressions.

from predefined shapes Some consoles offer libraries of common movement patterns, sometimes offering shapes that are not definable in the function language.

from freehand drawing Finally, operators can construct shape effects by freehand drawing, using a touchscreen or pointing device. Sophisticated consoles even allow the freehand construction and editing of curves, offering a subset of the functionality of common vector-graphics programs

5.7.5 Conflict Resolution

Consoles which support parallel playback or simultaneous playback and manual control hold the possibility of *conflict*: several sources of parameter information might request different parameter settings for the same fixture. In case two sources request control over a common parameter setting,, the console must resolve the conflict and decide on the actual parameter setting to send to the fixture. Three different conflict resolution strategies exist:

HTP HTP is for *highest takes precedence* and is suitable mostly for intensity settings: of two conflicting settings, the highest wins; the console transmits the maximum of the two.

LTP LTP is for *latest takes precedence*. LTP takes into an account the order in which the masters become active. Later masters take precedence over earlier ones.

Priority Another approach is to assign priorities to the various sources, or to allow the operator to assign them.

5.7.6 Stage Modelling

Operators identify fixtures by one of two means: by function or by position. The control protocols between console and fixtures identify a fixture by an artificial number. Therefore, operators usually keep a plan on paper with the fixture positions and these numbers drawn in. Some consoles maintain such a plan internally, and allow the operator to select fixtures by position directly. In fact, some manufacturers will, for a specific stage, custom-manufacture a console with positional controls.

Some consoles even maintain a three-dimensional model of the stage, along with more accurate physical modelling for the fixtures themselves, and their orientation in space. Moreover, these consoles can display the arrangement as well as the light beams itself. This allows the designer to get a rough impression of the lighting arrangements offline. It can help determining beam size and overlap, and avoid unfavorable arrangements of the fixtures.

5.7.7 Playback and Playback Storage

Fader boards The simplest consoles have no storage capability whatsoever, and only a single array of faders. Thus, the operator has to operate each fader manually during a fade.

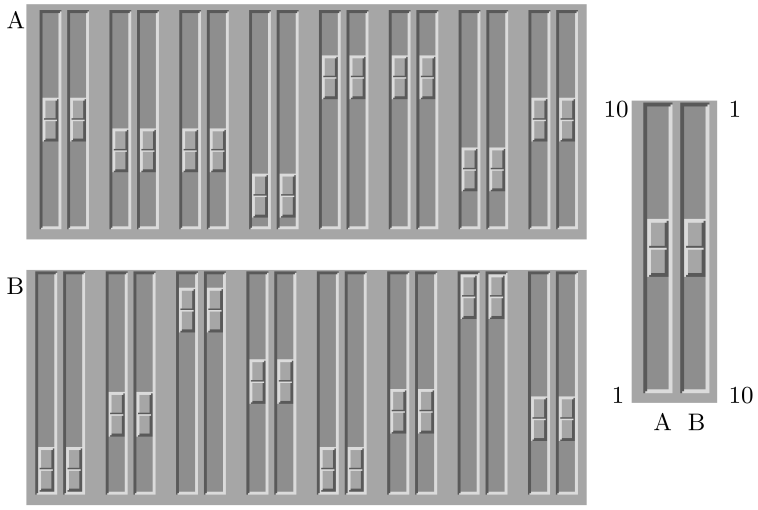

Figure 5.6: A rudimentary preset console.

Preset Consoles The next step up from the simple fader board is the *preset console*. It still has no storage, but it has two identical fader boards. Moreover, each board has an associated master fader which controls the relative intensity of the look "preset" into its board. Figure 5.6 shows the arrangement of a preset console. With, say, board A at 100% intensity, and board B at 0, the operator can prepare the target look for a fade on board B. The two master faders work in opposite ways. Thus, by pulling down both faders simultaneously, the operator effects a crossfade from the look on board A to the look on board B.

Sequential memory A sequential-memory console stores a *cuelist*—a sequence of looks with associated fade times. Each cue receives a sequence number, and the operator can trigger the fades in the cuelist by pressing a "Go" button. The consoles differ in the range of fade options available (see Section 5.3): some only offer crossfades, some do not offer delayed fades.

Multiple cuelists More sophisticated consoles maintain multiple cuelists for parallel playback. Multiple cuelists usually coexist with facilities for chases and effects.

Metainformation Finally, some consoles come with a regular character keyboard and allow the storage of comments along with the looks of a sequence. The

console displays them before activation to help the operator coordinate with the script or the action on stage.

5.7.8 Sequence Assembly and Parallel Playback

Consoles generally build on one of two principles for supporting the assembly of sequences of lighting events and playing them back later:

Cuelists and playback masters A cuelists is, as discussed above, a sequence of fades, occasionally annotated with additional ordering information for out-of-sequence playback or looping. Before playback, the operator assigns a cue list to a *playback master*, a collection of controls for playing back the events of the cue list. These controls usually include a "Go" button, two faders for controlling in and out fade position, an intensity-limiting fader, suspend/resume buttons, possibly faders for slowing down or speeding up fades. Multiple playback masters allow simultaneous playback of several independent cue lists.

Sequencer Sequencer consoles use an approach like an audio sequencer, with the fixtures being the conceptual tracks.

5.7.9 External Triggering

There are several common protocols which allow coordination between different control devices involved in a show; Section 3.6 contains an overview.

5.8 An Overview of Existing Consoles

This section provides an overview of the most popular high-end consoles. The cutoff criterion for including consoles in this selection was the support for both concerts and theater, support for multi-parameter moving lights as well as accurate modelling of different multi-parameter fixtures. The latter criterion excludes consoles which are historically dimmer-only consoles but have tacked-on support for multi-parameter fixtures—such as Rosco's Horizon [RoscoHorizon98,] or the Strand family of consoles [StrandGeniusPro, 2000, StrandTracker, 1998].

The first part of this section has short descriptions of all reviewed consoles, followed by tabular classifications of their look and effect subsystems. The classifications focus on the differences between the consoles.

5.8.1 Console Synopses

ASM R2-D2 The R2-D2 [ASMR2D2, 2000] uses a different design approach than the other consoles: the console consists of a handful of subsystems with priority-based conflict resolution. Instead of an array of playback masters, the show programming subsystem uses a sequencer. R2-D2 is the only console with priority-based conflict resolution, and thus does not have to deal with the complexity of LTP-based approach.

Avolites Sapphire 2000 and Diamond I/II The Avolites Sapphire and Diamond consoles [Mitchell, 1999, Marks, 1998] are tracking preset consoles with LTP conflict resolution.

ETC Obsession II The Obsession II console [ETCObsession, 1998] is, together with the WholeHog, one of the very few hierarchical consoles on the market. The Obsession calls presets "groups."

Flying Pig WholeHog II The WholeHog [Fly, 1999] has long been one of the most popular consoles for moving lights on the market. It has accumulated one of the largest feature sets. In particular, it is one of only two hierarchical consoles on this list. WholeHog's presets are integrated into its palette engine.

MA GrandMA The GrandMA [MAGrandMA, 2000] is one of the youngest consoles on the market, building on and enhancing the concepts from MA's successful Scancommander series [MAScanCommander, 1996]. Its distinguishing feature is the use of motorfaders to allow easy look editing. Moreover, it features TFT touchscreens with a modern windowing user interface. Its effect engine is one of the most sophisticated available. In particular, it allows the construction of composite function shapes and freehand editing.

High End Status Cue High End's Status Cue [HighEndStatusCue, 1997] is a PC-based system: High End provides only the software and an add-on control board with a DMX512 interface. The operator hooks up a standard Windows-based PC to the control board. The software itself follows standard GUI windows conventions. Otherwise, the console is fairly standard.

Martin Case The Martin Case [MartinCase, 2000] family of consoles is primarily for disco and concert use. Its design focus is on fast live access to parameters and effects. Its distinguishing feature is its stage modelling capability—the operator can define a stage grid and tell the console where each fixture is located within the grid.

Vari-Lite Virtuoso The Virtuoso [VariLiteVirtuoso, 2000] is large console with an emphasis on accurate modelling of its environment: Its stage modelling subsystem actually maintains a three-dimensional model of the stage, and its extensive data on the correspondence between moving-light parameter settings and reality even allow it to provide a rough simulation of the shapes and target areas of the light beams.

5.8.2 Look Construction Subsystems

	XYZ	conceptual colors	tracking	hierarchical	LTP/ priority	stage modelling
R2-D2	No	Yes	No	No	Priority	Yes
Sapphire Diamond	Yes	Yes	Yes	No	LTP	Yes
Obsession	No	No	Yes	Yes	LTP	No
WholeHog	Yes	Yes	Yes	Yes	LTP	No
Status Cue	No	Yes	No	No	LTP	No
GrandMA	No	Yes	Yes	No	LTP	No
Case	No	Yes	Yes	No	LTP	Yes
Virtuoso	Yes	Yes	No	No	LTP	Yes

Figure 5.7: A classification of the look construction subsystems of some consoles.

Figure 5.7 shows a classification of the look construction subsystems of the reviewed consoles. All reviewed consoles are preset consoles (see Section 5.7.7), support HTP conflict resolution as well as the forming of groups and submasters. The criteria in which they actually differ are the following:

XYZ means the console can point a moving light at a specific spot on stage, specified by Cartesian three-dimensional coordinates.

Conceptual colors means that console allows specifying color parameters device-independently through RGB, CMY or HSV triples, or manufacturer codes.

Tracking means the console stores look within a sequence in a relative way—it records only changes between successive looks (see Section 5.7.3).

Hierarchical means the console supports hierarchical presets (see Section 5.7.2).

LTP/priority specifies the conlict resolution strategy. "LTP" means the console gives precedence to the most recently activated subsystem. "Priority" means that the subsystems have assigned, fixed priorities that determine precedence.

Stage modelling says whether the console maintains at least a two-dimensional model of the rigging setup, and allows the operator to select a fixture by pointing at its graphical location.

5.8.3 Effect Construction Subsystems

	functions	composite functions	predefined shapes	freehand drawing	freehand editing
R2-D2	No	No	No	No	Yes
Sapphire Diamond	No	No	Yes	No	No
Obsession	No	No	No	No	No
WholeHog	Yes	No	Yes	Yes	Yes
Status Cue	No	No	No	No	No
GrandMA	Yes	Yes	Yes	Yes	Yes
Case	Yes	No	Yes	No	No
Virtuoso	No	No	No	No	No

Figure 5.8: A classification of the effect engines of some consoles.

Figure 5.8 is a classification of the effect engines (see Section 5.7.4. All consoles reviewed support chases as well as fade sequence effects. Here are the criteria for which the consoles actually differ:

functions means the console allows assigning mathematical functions over time to parameters.

composite functions means the operator can construct new functions from arithmetic expressions.

predefined shapes means the console has a library of geometric shapes and other shape effects.

freehand drawing means the operator can create a new shape by drawing it with a 2D pointing device.

freehand editing means the operator can interactively create and edit new shapes from lines and splines.

Part III

A Tour of Lula

Take a bite of Lula.
—Lula in *Wild at Heart*

This part of the dissertation gives a tour of Lula from the user's viewpoint. Lula is a large and complex application, but its basic concepts are simple and easy to master.

The two chapters in this part are not meant to be a manual. Rather, they demonstrate the important features of the system, especially in areas where Lula departs from the design practice of existing consoles. Where possible, the tour compares Lula with existing console. Mostly however, the conceptual differences are too great for such an endeavor to make any sense. Those areas which are indeed similar most often do not warrant an examination in this context.

Some of the most innovative aspects of the system are covered in depth in later parts of the dissertation—specifically, the cue model and the subsystem for animated lighting. The tour merely skims over those. A number of novel aspects of Lula not central to Lula's main structural innovations—dynamic dimmer curves, multi-section playback, and the priority system, for instance—were also swept under the rug.

Chapter 6 explains the basic functionality of Lula necessary to create simple shows with a linear sequence of light fades, using theatrical fixtures. Chapter 7 highlights some of Lula's advanced concepts, especially those related to animated light, parallel playback, and multi-parameter fixtures.

Chapter 6

Basic Lula

Hey, Tommy ... What's goin' on over
there in number four where al them
bright lights are all the time?
—Buddy in *Wild at Heart*

In theater lighting, an operator needs the lighting console to do three basic jobs: construct looks, assemble those looks into a sequence of light fades, and play back that sequence during the show. All of these tasks are straightforward in Lula. However, they differ significantly from the corresponding features of conventional consoles: look construction is *hierarchical* and allows the construction of looks from components, so-called *cues*. For sequence assembly, Lula contains a basic word processor; the operator pastes light fades into a script which may contain hints or even the text of the entire show. Finally, playback closely cooperates with the script editor, highlighting upcoming events and scrolling automatically.

6.1 Startup

As Lula starts up, it presents the user with four windows: the *script window*, a *cue editor*, a *fixtures view*, and a *sources view*. Figure 6.1 shows the arrangement. The script window is for assembling the light fades into a show. The cue editor allows the construction of look components and looks. The fixtures view shows the current parameter settings of the fixtures known to the system.

The fixtures view is initially empty; it fills up after *patching*—making the available fixtures known to the system. Patching in Lula does not differ significantly from other consoles, so a description of the process is omitted here. The sources view, finally, shows all windows which actively contribute fixture parameter settings. The color of the window icons corresponds to the colors displayed in the sources view (not visible in the figure); clicking on a button in the sources window raises the associated cue.

6.2 Constructing Simple Cues

The first job at hand is the construction of looks. For demonstration purposes, assume a scene involving the groundplan described in Section 4.5: the scene involves the sofa and the two chairs as playing areas. Moreover, some additional blue wash lighting is supposed to create a nighttime atmosphere ready for romance.

In Lula, the operator constructs looks from components, so-called *cues*. A cue specifies—at least for the purposes of this section—some fixtures and their intensi-

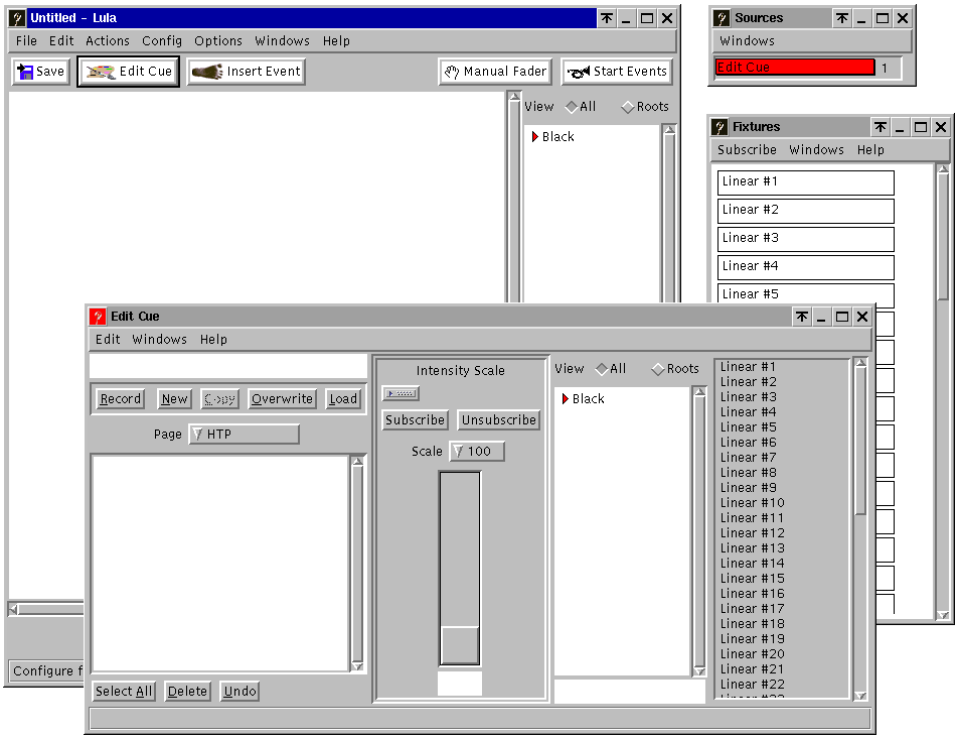

Figure 6.1: Windows after startup.

ties. The simplest cues contain only a single fixture. Compound cues result from combining several smaller cues. Each cue has a unique name chosen by the user. The window in front of Figure 6.1 is a *cue editor*, the tool for constructing cues. The blank area on the left shows the subcomponents of the current cue—the so-called *subcues*. The slider in the middle allows adjusting the intensity of subcues. The two areas on the right show the currently available cues and fixtures. One cue is predefined: `Black` is a cue specifying complete darkness.

Returning to the example setup, assume two fixtures trained on each of the chairs, one from each side, as well as three fixtures on the sofa: one from the left, one from the right, and from the front. Moreover, two blue floods provide the nighttime atmosphere. This corresponds to the partial rigging plan in Figure 4.14. Usually, the first job is to create a cue for each of the fixtures, giving it a name reminiscent of the function of its fixture. Figure 6.2 shows an example. The user can add subcues to the cue by double-clicking on one of the lists on the right. Lula determines parameter settings of the compound cue by HTP-combining those of the subcues (see Section 5.7.5). It does, however, support other of combination, as shown in the next chapter. In any case, the slider in the middle is for adjusting the relative intensities of the subcues.

Once this first phase is completed, the cue list will contain cues `Sofa Left`, `Sofa Right`, `Sofa Front`, `US Chair Left`, `US Chair Right`, `DS Chair Left`, `DS Chair, Right`, `Blue Flood Left` and `Blue Flood Right`.

Now all is set for the construction of larger cues: a cue `Sofa` contains three components: `Sofa Left`, `Sofa Right` and `Sofa Front`. When the user double-clicks on them in the list on the right-hand side, they appear in the editor on the left. By selecting single subcues and operating the slider, she can change their relative intensities. Note that the user, for constructing the `Sofa` cue, need not remember

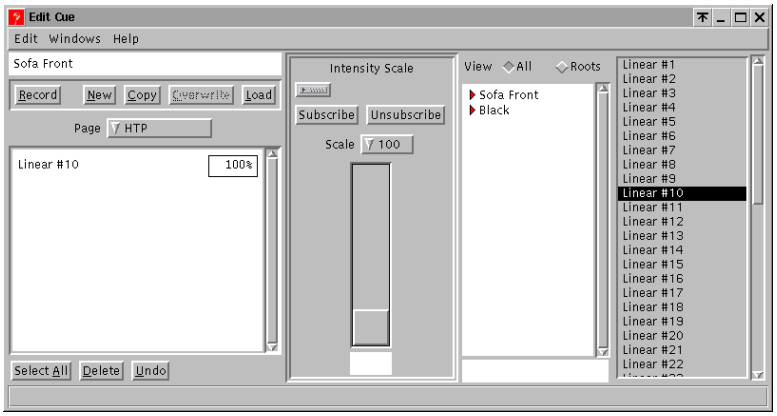

Figure 6.2: Cue editor with simple cue.

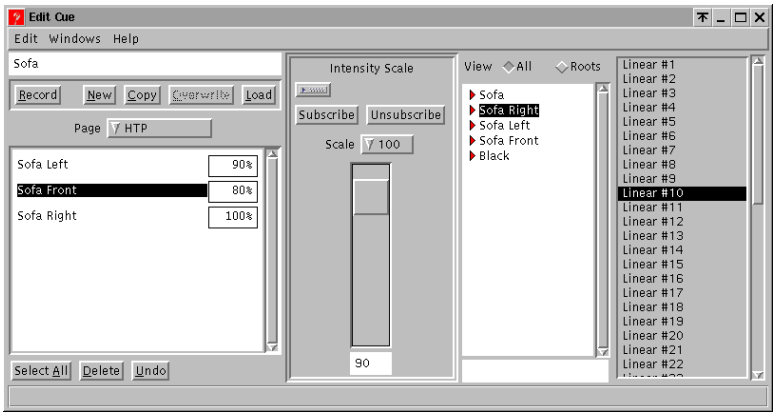

Figure 6.3: Cue editor with compound cue.

the particular fixtures or their numbers trained on the sofa—that information is hidden away in `Sofa`'s subcues.

Presumably, the user repeats the process for the two chairs, resulting in cues called `US Chair` and `DS Chair`, and for the two floods, resulting in a cue `Nighttime`. The next step is the construction of the cue for the basic lighting for the furniture—a compound cue with `Sofa`, `US Chair`, and `DS Chair` as subcues. Figure 6.4 shows the result.

Finally, `Furniture` and `Nighttime` combine to form the cue `Romance` which constitutes the look of the scene. `Romance` is ready for use in the show.

The cue list on the right-hand side of the cue editor allows the user to view the structure of the cue hierarchy in tree form by clicking the red triangles. Figure 6.6 shows the result after unfolding the `Sofa` cue in this way.

6.3 Modifying Cues

The user can load a cue back into the cue editor at any time by selecting it in the cue list ands pressing the `Load` button. All changes made to a cue immediately propagate to all other containing it.

In the example, assume the fixture contained in `US Chair Left` needs to be

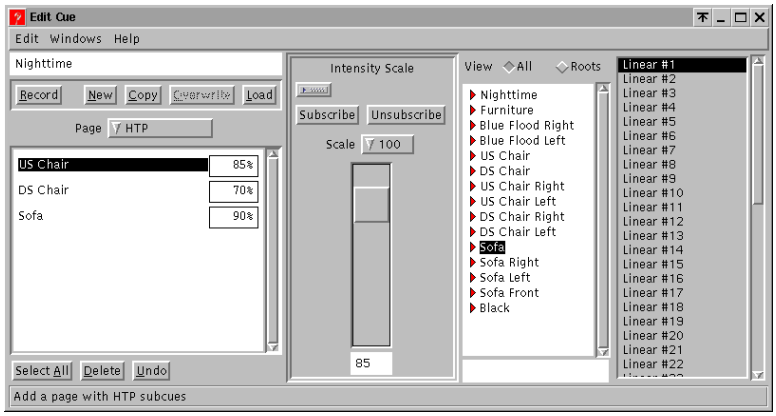

Figure 6.4: Cue editor with compound cue.

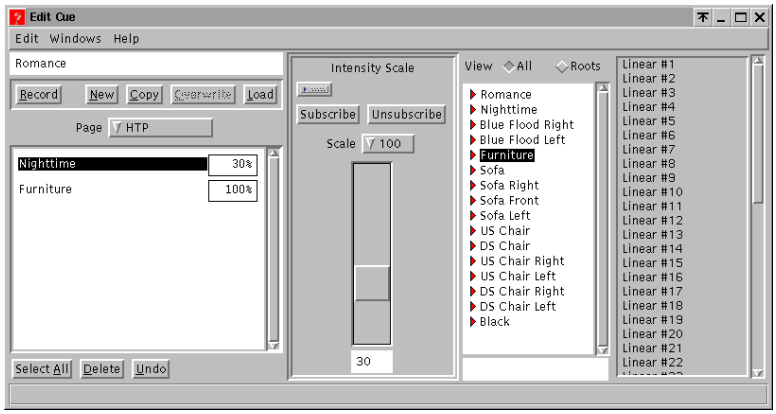

Figure 6.5: Cue editor with look cue.

softer everywhere it occurs, possibly because of a mistake or because the fixture needed to be replaced by a stronger one. When the user turns down the intensity of the fixture in `US Chair Left`, all cues containing `US Chair Left` will reflect the decreased intensity immediately. The affected cues are `US Chair`, `Chair`, `Furniture`, and `Romance`. The user can see the effects of changes to a subcue in a cue containing it live by opening a second cue editor to make the changes.

Thus, the components of a cue are really only *references* to other cues which makes Lula a *hierarchical* console (see Chapter 5). This distinguishes Lula from most other consoles which are conceptually flat. It greatly simplifies making pervasive changes, a frequent task in practical lighting design.

Lula's cue system requires proper use to be effectively: the cue hierarchy must correspond to the conceptual model of the lighting looks. This means creating cues corresponding to their conceptual function in the show on stage rather than their implementation through particular fixtures. Specifically, the user needs to exercise some care when sharing subcues between cues: this is only appropriate when the subcues fulfill equivalent functions. Otherwise, a change to the subcue might lead to the desired effect in cue which contains it, but do something unwanted in another.

Figure 6.6: Cue hierarchy display.

6.4 Assembling the Script

With a single look in place, Lula is now ready to accept the programming of a script. Lula's main window is called the *script window*. The blank area in the middle is a simple word processor: it allows typing in text or pasting it in from other applications. The editor supports simple attributes such as font size, font family and color. Moreover, the script can also contain pictures.

Assume the scene of the example is part of a simple play: the show starts out in blackout. Actors come in and sit down on the furniture. The lights come up on the `Romance` look, and the show starts. After some time, the scene ends and the lights fade to black.

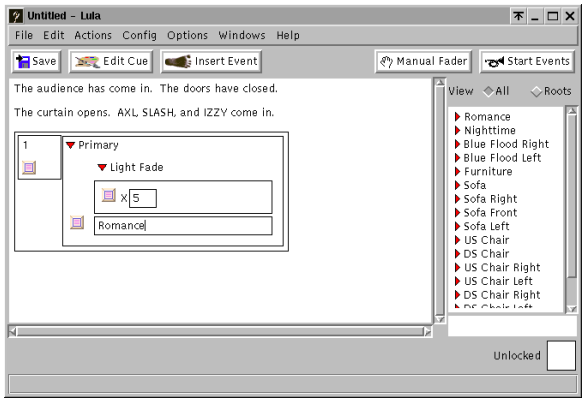

Figure 6.7: Light fade in a script.

The scripts starts off with some introductory remarks. The user can insert an instruction for a light change by pressing the button labeled `Insert Event`. A `Light Fade` box appears, by default set to a crossfade, marked by an `X`. The user must complete the blank fields for the fade time and the name of the target cue, respectively. Figure 6.7 shows the result. The menu button to the left of the `X` allows switching to other fade types such as in/out fades, or in/out fades with delays (see Section 5.3).

Presumably, the dramaturgy department has provided electronic text of the conversation taking place in the scene. The user can paste that text into the script and insert another light-fade event after the end of the conversation. Figure 6.8 shows the final portion of the script. The script is now ready for playback.

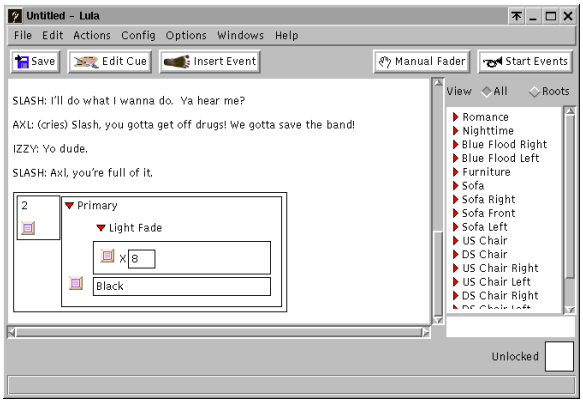

Figure 6.8: Final light fade.

6.5 Playback

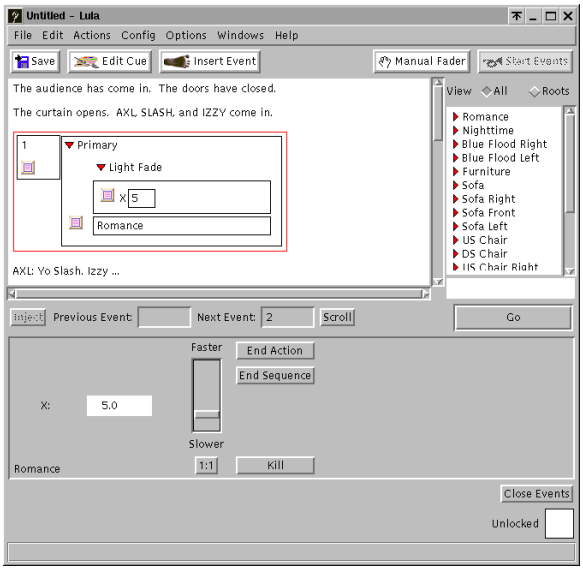

Figure 6.9: Start of playback.

The operator starts playback by pressing the `Start Events` toolbar button: the script window splits as shown in Figure 6.9, showing the script as well as the *playback panel* at the bottom.

The lower part displays what actions are waiting to happen or running. At the onset of playback, the stage is still dark, and Lula waits for a signal to start the first light fade. The script highlights the upcoming event with a red border. At any time, the user can press the `Scroll` button to move the next upcoming event into the script window.

The user starts the first event by pressing the `Go` button: the fade time display shows the progress of the fade. Moreover, the user can intervene in the running fade with the speed slider, thereby making the fade go faster or slower. The `End Action` and `End Sequence` buttons end the light fade prematurely, moving on the the next.

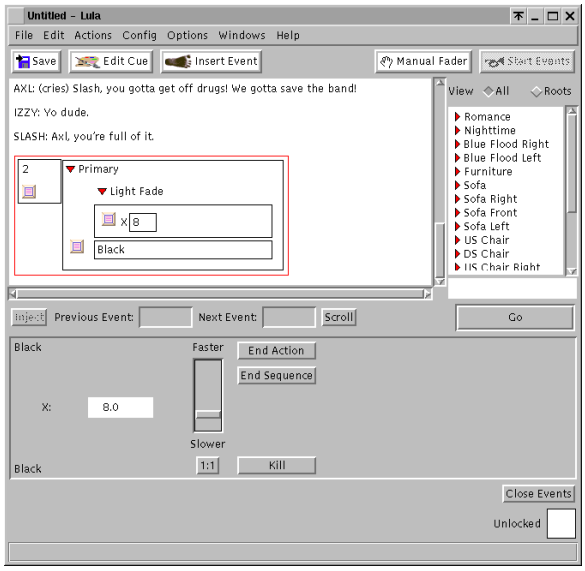

Figure 6.10: Playback.

After the first fade has completed, the next event moves into the playback panel. By pressing `Scroll`, the user can also get the script to display the relevant part containing the event. Figure 6.10 shows the situation.

At any time, the user can change the sequence of events by clicking on an event in the script window: a menu with a single entry labeled `Inject` appears. Selection causes the event to become the next event in the playback panel. Figure 6.11 shows the menu.

6.6 Manual Control

A number of situations require direct manual control over the look on stage. Examples include lighting the curtain call, reacting to spontaneity on change, or addressing emergencies. For this purpose, pressing the `Manual Fader` button pops up a manual fader window. Double-clicking on a cue produces a slider which the user can activate. Once activated, operating the slider brings up the cue. Figure 6.12 shows a manual fader with a slider for the `Sofa` cue.

6.7 Changing Venue

Once the evening's show is over, the production could tour to a different location. Given the time constraints usually associated with a guest performance—usually only one day for the complete setup, often less—it is important that re-programming the lighting does not take long.

From venue to venue, the structure of the show will not change, and neither will the basic layout of the set and the basic structure of the lighting. If the cue structure reflects the structure of the lighting, adapting to a new venue is straightforward: Only the "leaf cues" `Sofa Left`, `Sofa Right`, `Sofa Front`, `US Chair Left`, `US Chair Right`, `DS Chair Left`, `DS Chair Right`, `Blue Flood Left` and `Blue Flood Right` contain references to actual fixtures. Thus, it is necessary to change these cues to refer to the fixtures in the new setting. Consequently, the new cues

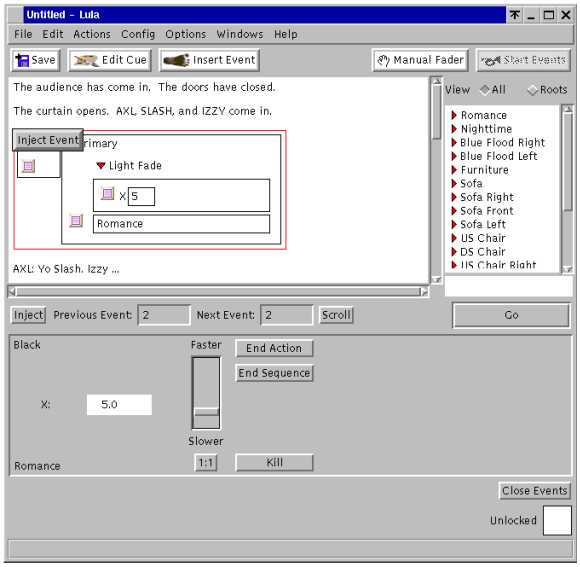

Figure 6.11: Injecting an event.

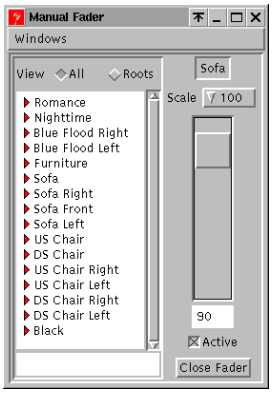

Figure 6.12: Manual fader.

reflect the changes in the lighting hardware, no more and no less. Possibly, the distribution of fixtures is different: there may be only two fixtures trained on the sofa, or three fixtures on one of the chairs, for example. In this case, it is necessary to change the cues at the level above. Note that this still merely reflects the changes in hardware.

The rest of the cue structure, as well as the script, is preserved throughout these changes. In all probability, the new lighting is already functional at this stage. Further adjustments are necessary to achieve optimal looks, but these are typically minor.

Chapter 7

Advanced Lula

Let's go dancin', peanut. I'm ready.
—Sailor in *Wild at Heart*

The essential features described in the previous chapter are sufficient for many theatrical shows. Lula has a number of capabilities beyond those revealed in the previous chapter: Lula's cue editor, in addition to the standard HTP combination of subcues, supports combining cues via *restriction* and *difference* which considerably enhance the expressiveness of cues. Also, Lula supports dynamic lighting situations such as concerts with multi-parameter fixtures with its added demands on light animation and advanced playback. Cues can specify non-intensity parameters, thereby supporting multi-parameter fixtures. Moreover, Lula has extensive support for dynamic lighting through the use of animated light specified by *reactive programs* and advanced playback features through the use of *sequencers* and *parallelizers*.

7.1 Advanced Cue Operations

By default, combining several cues into a larger cue follows the HTP principle: Lula combines the parameter settings of the subcues; if several subcues specify intensities for a common fixture, the combined cue specifies their maximum. Thus, as a cue gathers more and more subcues, it can only get brighter.

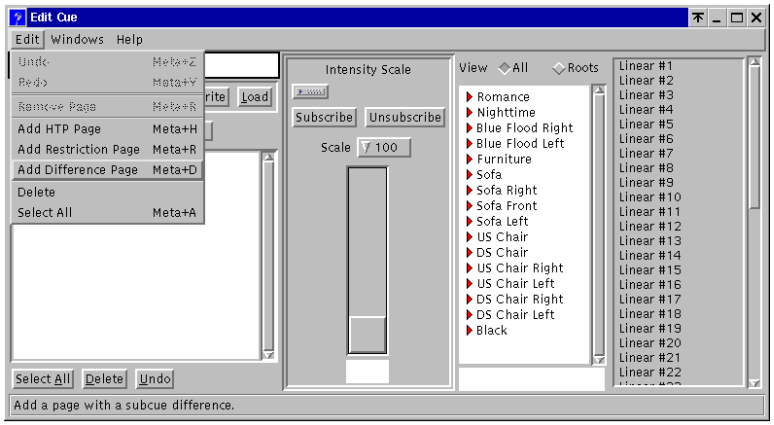

Figure 7.1: Adding a page to a cue.

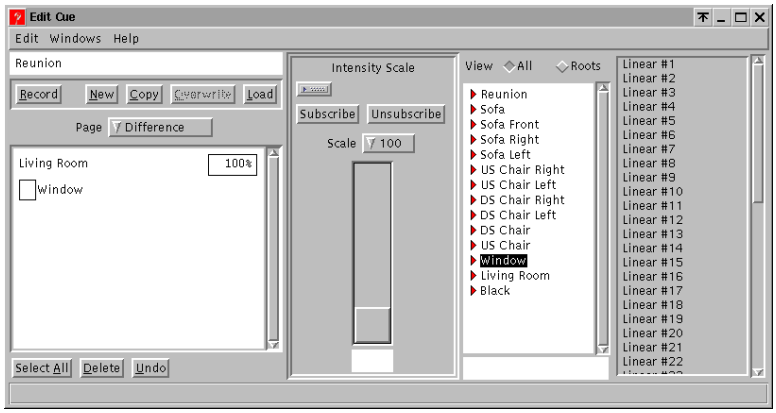

Figure 7.2: Cue editor displaying a difference page.

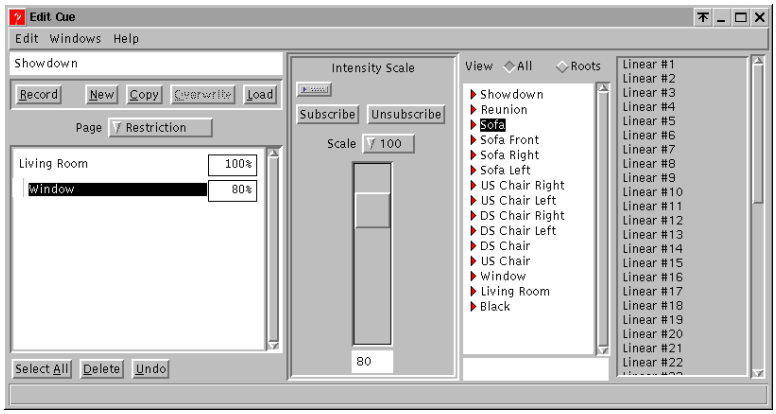

Figure 7.3: Cue editor displaying a restriction page.

Some natural cue constructions require means of combination different from HTP. For example, a typical look for the groundplan from Chapter 4 might be "base lighting for the entire living room without the window." The natural component cues for this look are for "base lighting" and for "window," yet HTP offers no way to remove the window from the base lighting cue. This cue construction is a an example for the *difference* between two cues.

Another cue construction not covered by HTP is "base lighting for the entire living room with the window dimmed down." Again, the natural component cues are "living room" and "window," but neither HTP nor difference allows this construction. Rather, this cue is a so-called *restriction* of the living room cue with the a dimmed-down version of the window. Restriction is equivalent to difference followed by HTP.

Lula offers difference and restriction. These means of combination are new—they are not available in any other lighting console. Chapter 10 systematically develops the requirements that lead to the inclusion of difference and restriction. Moreover, the three combinators are the basis for an algebraic characterization of cues which were instrumental in their design. Chapter 11 contains an extensive account.

Lula offers difference and restriction through the `Edit` menu: the user can add additional *pages* to the cue. Each page contains subcues combined either by HTP

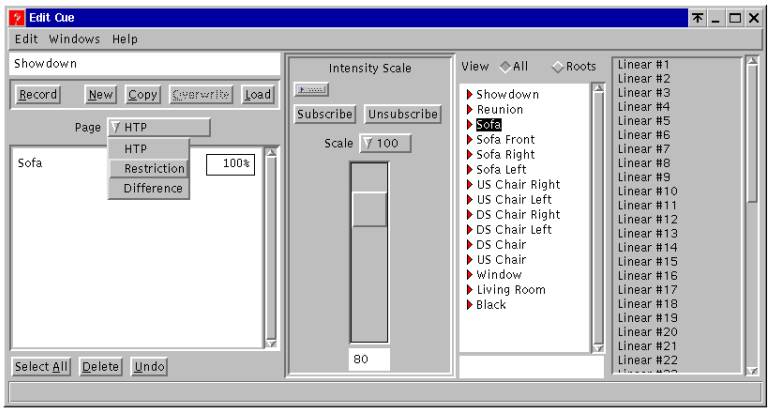

Figure 7.4: Selecting a page.

or by restriction or difference Figure 7.1 shows the relevant menu choices. Lula combines the pages constituting a cue by HTP. In practice, however, most cues consist of just a single page. Initially, a cue contains a single HTP page.

Figure 7.2 shows a cue editor displaying a difference page: the base subcue is at the top; the other subcues are "subtracted" from the cue. By clicking on the box to the left of one of the subtracted subcues, the user can specify the *complement* of that cue—all fixtures *not* included in it. Subtracting the complement of a cue acts rather like a stencil: the difference of a cue a and the complement of another cue b specifies all fixtures which a and b in common at the intensities specified in a.

Figure 7.3 shows a cue editor displaying a restriction page: the subcue at the top is restricted by all subcues under it.

Figure 7.4 shows how the user can switch between the pages constituting a cue by clicking on the choice button.

7.2 Non-Intensity Parameters

Lula has full support for multi-parameter fixtures. Setting non-intensity parameters is similar to setting intensity: in the middle of a cue editor, various controls for the other parameters pop up. Figure 7.5 shows a cue editor with a visible pan/tilt control.

By default, operating the control has no effect on the selected subcues. Rather, subcues must *subscribe* to the control. The user subscribes a subcue to a control by pressing the `Subscribe` button. Subsequently, a display of the parameter setting shows up next to the subcue. Figure 7.6 shows such a situation.

Just like with intensity, non-intensity parameter settings apply uniformly to all fixtures contained in a subcue. The effect of a pan/tilt setting differs from that of the intensity scale: The intensity slider is a *relative* control—it merely multiplies the intensities specified in a subcue by some factor. In contrast, the pan/tilt control is *absolute*—all fixtures in the subcue simply assume the settings specified by the control, potentially overriding settings specified by the subcue itself.

A number of other controls are available, corresponding to the most common parameters available in modern multi-parameter fixtures with remote control (see Chapter 2). For a more systematic description of how parameter settings work, see Chapters 10 and 11.

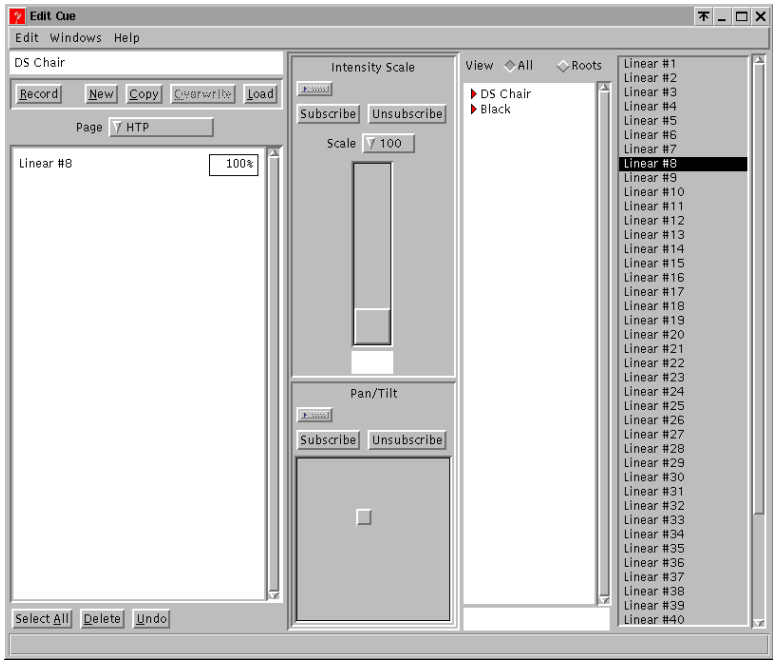

Figure 7.5: Cue editor with pan/tilt control.

7.3 Animated Lighting

So far, light fades have restricted playback to linear transitions between fixed looks. In addition, Lula can also animate cues—continuously change parameters according to the specifications of the operator. This applies to theatrical settings—a "breathing" look, simple chases, strobe effects, and so on. Animation is especially attractive with multi-parameter fixtures—tracing geometric figures with the light beams, fans, ballyhoos, etc.

To this end, Lula includes a small programming language called *Lulal* which is the basis for a library of reactive effects. In Lulal, the user specifies *tasks* composed from reactive *behaviors* and *events*: A behavior is a specification of a value changing continuously over a time. It can react to external events—alarm timers, buttons and other controls, or external hardware triggers. A task finally specifies the execution of a behavior with a beginning and an end. Chapter 12 has full details on the Lulal language.

Figure 7.7 shows the Lulal editor window, a modest interactive program environment for Lulal. Prior to playback, Lulal performs static syntax and type checks to avoid the most common programming errors. The `Validate` buttons performs these checks; validation highlights erroneous expressions. Figure 7.8 shows an example.

With an effect library in place, the user can specify effects as actions in the script window, either directly naming a task specified in the Lulal window, or calling a procedure defined there to return that task. Figure 7.9 shows an example. Lula as shipped includes an effect library written in Lulal ready for inclusion in scripts: the user does not have to learn Lulal to create effects. Only for more advanced effects, some programming is unavoidable. In turn, the user gains expressive power other consoles do not offer.

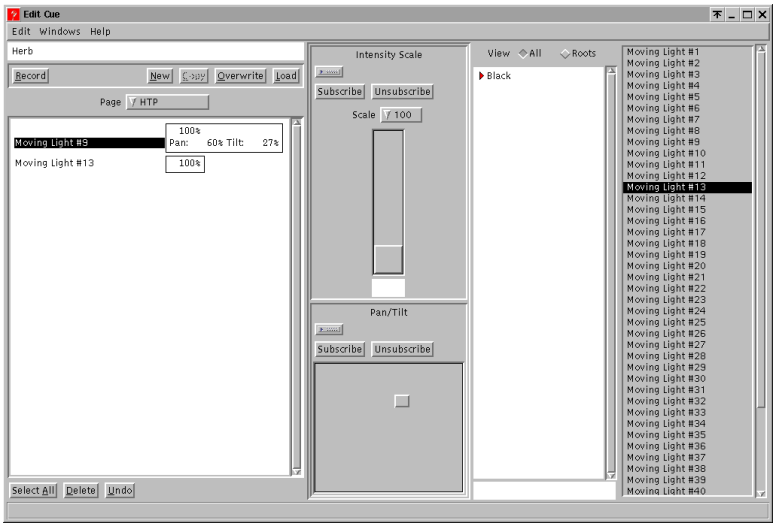

Figure 7.6: Subscribing to pan/tilt control.

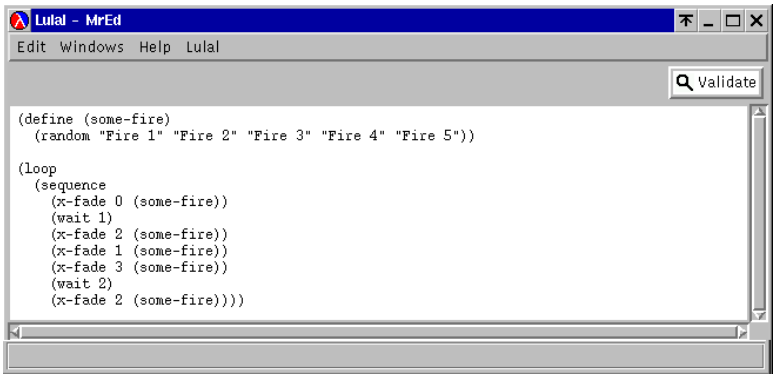

Figure 7.7: Lulal window with Lulal program.

7.4 Advanced Playback

Playback is not restricted to a linear succession of actions. Rather, the user can customize the playback engine through the addition of *playback masters*. Each playback master executes functions independently from the others. This allows the playback of multiple parallel strands of activity.

Lula offers two kinds of playback masters:

Sequencers A sequencer performs actions sequentially. The primary display in the playback panel is a sequencer. When the user injects new actions into a sequencer, it will flush any pending actions.

Parallelizer A parallelizer contains multiple sequencers. Whenever a new action arrives, the parallelizer creates a new sequencer to perform it. When an action ends, the parallelizer deletes the corresponding sequencer again.

The user can add an arbitrary number of playback masters. Each playback master has a name for reference in the script window. Figure 7.10 shows the relevant dialog.

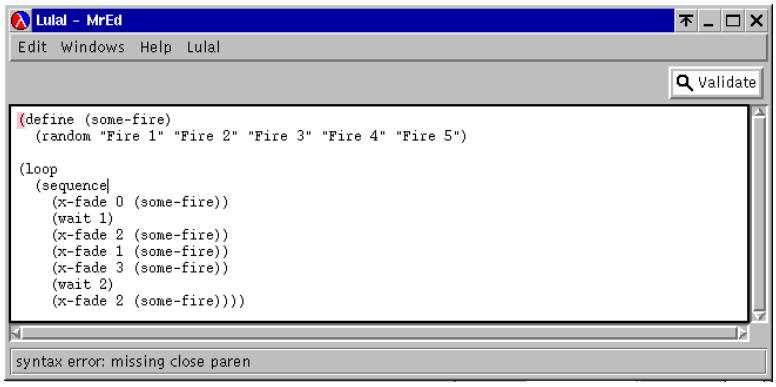

Figure 7.8: Lulal window with syntax error

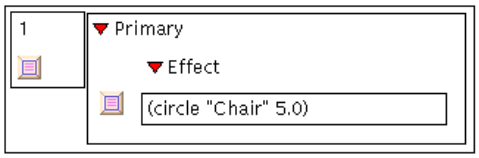

Figure 7.9: Effect event in script window.

(Other consoles usually offer only a fixed number of sequencers and no parallelizers at all.)

In the script window, the user can add so-called *secondary actions* to an event which address playback masters other than the primary ones. Upon playback, Lula will inject these secondary actions into the specified playback master. Figure 7.11 illustrates the selection of a playback master for a secondary event.

Upon playback, Lula displays a separate panel for each playback master. Each new arriving event injects its actions into the respective playback masters. The `Go` button applies to all playback masters collectively. However, stopping or killing off the actions in a playback master is possible on an individual basis.

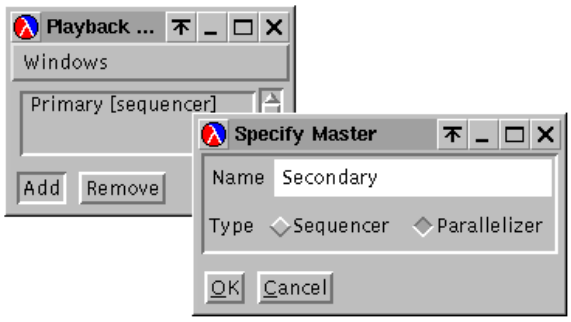

Figure 7.10: Creating new playback masters.

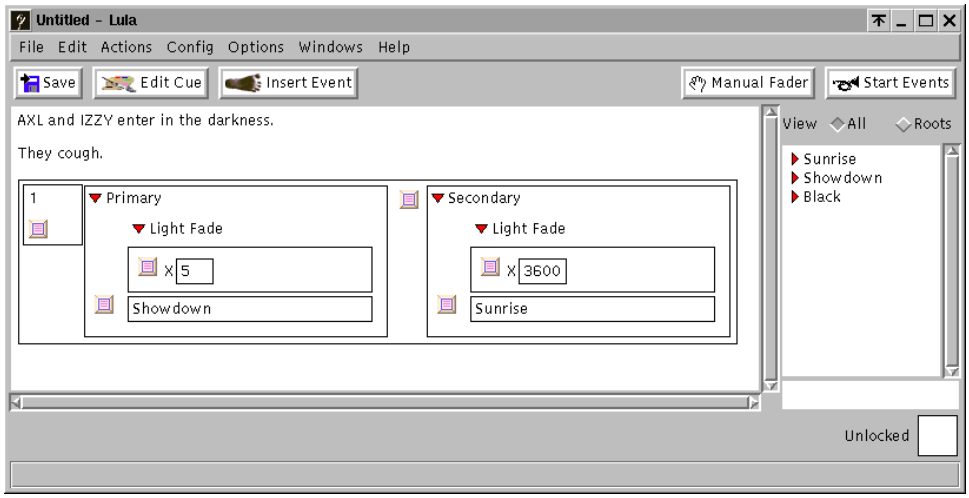

Figure 7.11: Adding secondary actions.

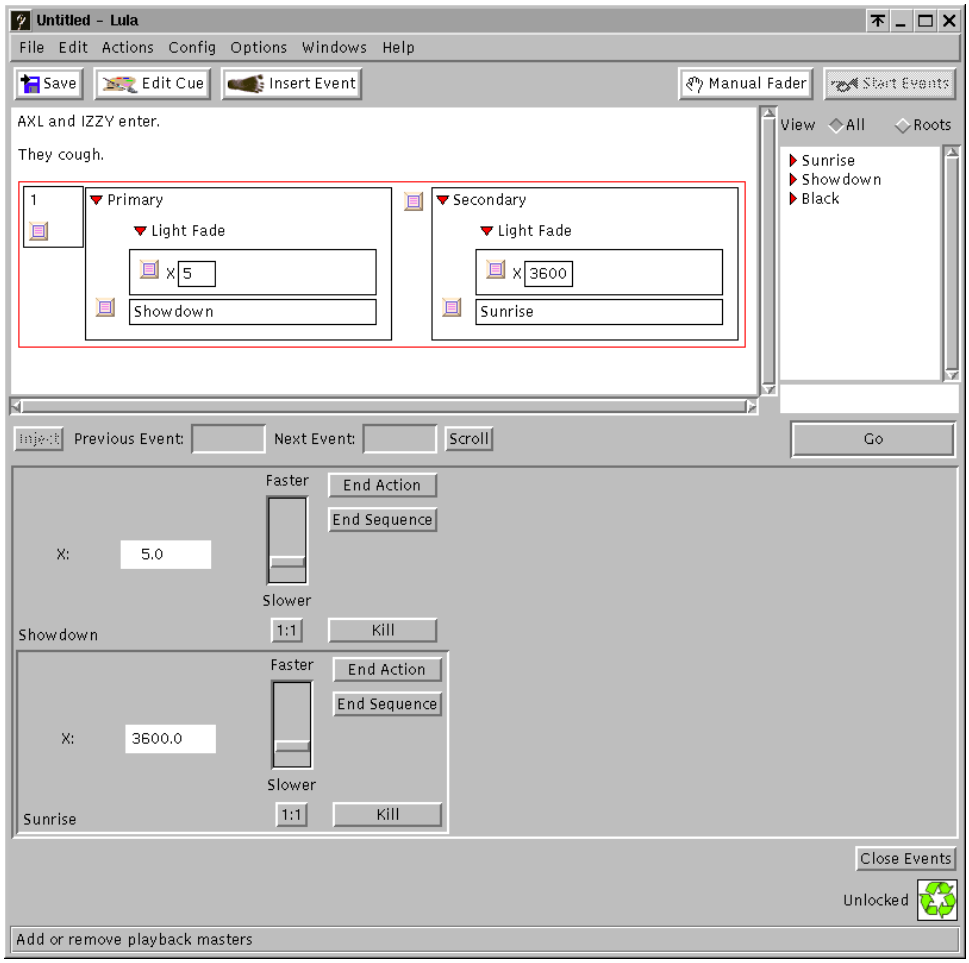

Figure 7.12: Playback of secondary actions.

Part IV

Lula Architecture

Hard to tell what's shakin' in a
place like this, honey. You don't
want to be walkin' in the wrong door.
—Sailor in *Wild at Heart*

The development of a piece of application software starts with some important choices that determine the structure of the final products in important ways. These choices broadly fall in two main areas:

Tool Selection Ultimately, the application will consist of large amounts of code. A number of tools are responsible for helping the developers produce that code. In some cases, integrated development environments and CASE tools might play an important part. However, by far the most crucial tool choice is that for a particular programming language, as orders of magnitude lie between the effectiveness of languages used today. With the choice of programming language comes the choice of programming paradigms to use, the choice of libraries which can benefit the application, and the choice of the implementation of the language.

System Architecture With the tools in place comes a broad subdivision of the task at hands into parts along with designs of the communication pathways and dependencies among them. The choice of these architectural components again strongly depends on the choice of programming tools in the first phase.

This part of the dissertation explains some of the architectural decisions which have shaped the Lula code. The first chapter—Chapter 8—indeed concerns the tools and techniques used which are largely independent of the application at hand. The second chapter of this part—9—is a structural overview of the Lula code.

Chapter 8

Tools and Techniques

I think they are too serious, these American fishermen. In Honduras, we are not so concerned with the method.
—Reggie in *Wild at Heart*

The choice of software development tools and techniques have shaped Lula in important ways, both structurally and in the functionality available to the end user. The pivotal choice is that of the programming language: Lula is written in Scheme, and with Scheme comes a wide assortment of programming paradigms and techniques available to the software developer. This chapter gives a brief overview of the techniques that play prominent roles in the Lula source code.

8.1 Scheme

Scheme [Kelsey *et al.*, 1998], seen by many as a descendant of the Lisp family of languages, is related to Lisp only in its similarity of syntax. Semantically, Scheme has its roots in the λ calculus [Barendregt, 1990] and the Algol family of languages[1]. Scheme has mostly become popular as a language for teaching introductory computer science [Abelson *et al.*, 1996, Hailperin *et al.*, 1999] and programming language research. Only in recent years, Scheme implementations with dedicated support for application development have emerged.

Here are the most important properties of Scheme as they apply to day-to-day programming:

- Scheme's syntax is extremely regular and simple which makes it easy to remember.
- The core language is very small, again reducing space requirements in the head of the programmer.
- Scheme mandates *automatic memory management* or *garbage collection*.
- Scheme is *higher-order*: procedures are first-class values. This, among other things, allows the construction of special-purpose "glue" to connect program components, define new control structures, and object systems. Section 8.4 below has more on the matter.
- Scheme is *extensible*: user-defined procedures look exactly like the primitives provided by the language.

[1] In fact, the fourth revision of the definition of Scheme bore a dedication to Algol 60.

- The language supports, in addition to abstraction over values, *syntactic abstraction* through a hygienic macro system. Again, user-defined syntactic forms look exactly like the built-in ones.
- Scheme has *latent* or *dynamic typing*: types are attached to values at run time rather than to variables at compilation time. All procedures are fully polymorphic.
- Scheme has *first-class continuations*; a program can capture the program execution context and restore it at any given time. First-class continuations are extremely useful for implementing non-local controls structures such as exception handling, coroutines, concurrency, and non-determinism.

The unifying upshot of these aspects is that Scheme basically allows the programmer to abstract over *everything*: „ordinary“ first-order values, procedures, control, and syntax. This allows the programmer to implement libraries which enable entire programming paradigms (this chapter lists a few below), thereby effectively defining new languages.

The resulting style of programming—identifying the paradigms needed for a particular application domain, implementing those paradigms as a library, the using them—originated with Lisp [Steele Jr. and Gabriel, 1993]. Scheme's flexibility enables this to the extreme. It also explains its popularity in teaching: computer science educators typically use the language merely as a vehicle for teaching paradigms rather than putting the language itself at the center of a course. Moreover, it explains why Scheme has survived despite its having been a niche language for so long [Steele, Jr., 1998].

Note that this „little language“ approach to programming runs somewhat contrary to recently popularized notion of using *patterns* [Gamma *et al.*, 1995]: patterns are generally extralinguistic natural-language instructions for solving recurring programming problems. Scheme programmers generally implement patterns as code, thereby obviating the concept, or depending on the viewpoint, having used it for decades before the term emerged [Chambers *et al.*, 2000]. This chapter shows some examples.

8.2 DrScheme

The Lula code runs on a *DrScheme* [Felleisen *et al.*, 1998, Flatt *et al.*, 1999], a Scheme implementation developed at Rice University as part of their Educational Infrastructure program. Originally targeted at educational software, its main focus is on providing an interactive programming environment suitable for beginning students in computer science. Thus, the project involves the creation of graphical user interfaces, and the safe interactive execution of Scheme programs within the DrScheme environment.

Because of those extensive requirements, DrScheme's Scheme engine contains functionality more often associated with operating systems [Flatt *et al.*, 1999]:

- concurrent threads of execution,
- GUI context management, and
- resource control.

Besides these, DrScheme offers a number of other facilities instrumental for the development of Lula:

- a GUI framework portable between X Windows, Microsoft Windows, and MacOS [Flatt and Findler, 2000b, Flatt and Findler, 2000a],

- a collection of GUI widgets for multimedia editors,
- a higher-order, fully-parameterized module system [Flatt and Felleisen, 1998], and
- an object system with single inheritance, supporting parametric inheritance via *mixins* [Flatt *et al.*, 1998] (more on that below in Section 8.7),

8.3 Automatic Memory Management

Automatic memory management, often known under the name of its most popular implementation, *garbage collection*, has long been known to be crucial for fully modular programming [Wilson, 1992]. It is crucial to an application like Lula for two reasons:

- In lighting control software, reliability is paramount: the software might have to be up and running for a long time. Space leaks, as are almost unavoidable with application-controlled memory reclamation, are unacceptable. Of course, persistent space leaks are still possible with garbage collection, but much less likely and much easier to avoid, as there are few permanent data structures in Lula which might grow indefinitely over time.
- With garbage collection, the memory management subsystem has more control over how, when and where memory allocation happens as opposed to `malloc/free`-style manual management. This enables a garbage collector to be *incremental* which is important for Lula's (soft) real-time requirements.

8.4 Higher-Order Programming

The one single aspect of Scheme which makes it particularly effective for program development in the large is its support for *higher-order programming*—the use of procedures as first-class data values. In practice, Scheme programs contain many *higher-order procedures*: procedures that take other procedures as parameters or return procedures. Higher-order programming has many uses in practical programming. These are among the most important:

- custom-built local control structures
- glue for program construction
- effective data representation

Local Control Structures A simple typical example for a higher-order procedure is the built-in `map` procedure. `Map` takes as arguments a procedure with one parameter as well as a list; it applies that procedure to all the elements of the list and returns a list of the results:

```
> (map (lambda (x) (* x 10)) '(1 2 3 4 5))
(10 20 30 40 50)
```

`Map` is a small useful abstraction which generalizes over one very common form of loops. The most immediate effect is a reduction in program size.

The principle underlying `map` extends well beyond loops: any container-like data structure such as vectors, trees, tables, and so on, often induces a small number of loop forms used to traverse them. Higher-order procedures are a good means

of abstracting over those loops, comparable to iterators and the visitor pattern in object-oriented languages, albeit with less notational overhead.

Lula uses inductive data structures extensively, specifically for cues and presets (see Chapter 11). A number of higher-order procedures provide the control structures which iterate or fold over them.

Program Glue In traditional procedural languages, the only way to combine procedures is to call one procedure from another. A higher-order languages allows the construction of new forms of glue between program parts. A trivial example is the `compose` procedure which composes two procedures:

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g (x))))))
```

`F` and `g` must be procedures of one parameter; `compose` returns the composition `f` ∘ `g` of `f` and `g`.

Higher-order procedures are the enabling technology for some very effective large-scale program structuring techniques such as *monads* [Wadler, 1992] which is a basis of Lula's reactive programming substrate as described in 13.

Data Representation Procedures essentially allow the wrapping of computations in data structures, which in turns opens up new opportunities in data representation.

A typical example is *delayed evaluation*, replacing a value by a procedure which will compute it. Delayed evaluation allows structuring many intricate algorithmic problem in natural way. Lula in particular uses it extensively for the representation of reactive values as explained in Chapter 13.

Delayed evaluation coupled with memoization yields *lazy evaluation*, a particularly effective technique for expressing many normally state-based computations in a purely functional way as well as for formulating functional counterparts to many imperative algorithms [Okasaki, 1998]. This is why recent functional languages such as Haskell [Haskell98, 1998] have adopted lazy evaluation as their default evaluation strategy.

8.5 Functional Programming

Another aspect of Scheme is its support of *functional programming* or *purely functional programming*[2]. Functional programming is a programming style which avoids the use of assignment and other side effects. Procedures written in this style behave rather like mathematical functions: passed the same arguments, they will always return the same result because they do not use assignment to remember any information from one call to the next.

Since functional programming primarily means doing *without* a common programming construct, languages supporting functional programming must make up

[2]There is some disagreement about the use of the term. In many contexts, it actually refers to higher-order programming. This section uses it in the sense suggested by the `comp.lang.functional` FAQ:

> Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these language are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style.

for the loss by offering powerful means of abstraction not usually available in traditional languages:

- cheap general-purpose functional data structures such as lists and trees (as opposed to inherently imperative data structures such as arrays and hash tables)
- higher-order programming
- lazy evaluation
- automatic memory management

With these means in place, the designers of modern functional languages such as Haskell [Haskell98, 1998] have chosen to remove assignment from the language altogether. In the development process, the use of functional programming has a number of important advantages, among them:

equational reasoning In a functional program, meaning is not affected by substituting equals for equals, a property also called *referential transparency*. This gives the programmer considerable more power for reasoning about the behavior programs, both formally and mentally.

persistent data structures A purely functional program never modifies a data structure in place. Rather, when it needs a changed version of a data structure, it will create a new one while the old data structure remains in place using copying and sharing. This again reduces worry in the programmer's head: when a program is holding on to a data structure, the programmer need not worry that some other part of the program modifies it in unpredictable ways.

no need for synchronization Synchronization between concurrent processes operating on shared data always means synchronizing modifications to that data. When there no modifications, no synchronization is necessary.

In the development process of Lula, a number of central data structures which originally had imperative representations have successively been replaced by functional ones:

- Cues (see Chapter 11) used be to be modified in place whenever the user made a change in the cue editor. The new Lula simply generates a new one. This has reduced program complexity as well as locking issues significantly.
- Presets (again, see Chapter 11) used to be based on arrays. Numerous synchronization issues arose, and the original code had significant bugs. Moreover, the array is an inherently fixed-size data structure, and some size restriction in Lula 3 can only be changed by a restart of the program. The new preset representation is functional, based on lists, cutting down on program complexity and bug count.
- Actions (explained in more detail in the following chapter) used to carry state. This caused numerous race conditions and other bugs in the original code which appeared only under marginal conditions. The current code avoids this, again resulting in less and cleaner code, and less bugs.

8.6 Data-Directed Programming

Data-directed programming is a generic term for programming techniques for dealing with different data types which support a common interface. Consider a classic example for data-directed programming, polar and rectangular representations of complex numbers [Abelson *et al.*, 1996]. Both representations supports the same operations, but their implementations are different. The standard procedural solution to the problem is explicit dispatch on type:

```
(define (+-complex number-1 number-2)
  (if (complex-rectangular? number-1)
      (+-complex-rectangular number-1 number-2)
      (+-complex-polar number-1 number-2)))
```

This style of programming is tedious, and, more seriously, requires that all representations are known at the time of writing of `+-complex`. If there are many such generic operations, the programmer must modify all of them when she adds a new representation. This may not even be possible if the code is compiled.

The most popular technique for implementing data-directed programming is *message-passing style*: the programmer makes the operations for a particular representation part of the representation itself, and subsequently "sends the representation a message" to retrieve the operation. In Scheme, this is easily accomplished by using a procedure which accepts the message and performs the operation:

```
(define (make-complex-rectangular real imaginary)
  (lambda (message)
    (case message
      ((real-part) real)
      ((imaginary-part) imaginary))))

(define (make-complex-imaginary angle magnitude)
  (lambda (message)
    (case message
      ((real-part) (* magnitude (cos angle)))
      ((imaginary-part) (* magnitude (sin angle))))))

(define (+-complex number-1 number-2)
  (make-complex-rectangular
    (+ (number-1 'real-part) (number-2 'real-part))
    (+ (number-1 'imaginary-part) (number-2 'imaginary-part))))
```

In this code, all complex numbers—regardless of the representation—are represented as procedures accepting one of two messages, `real-part` or `imaginary-part`, yielding the real and imaginary component of the number, respectively. The rectangular representation uses simple selection, the polar representation computes the components—from the outside, they both look the same. The new `+-complex` procedure uses no type dispatch itself. This makes the operations on numbers easily extensible. Lula uses data-directed programming in most of the places where different representations support the same interface:

General behaviors Lula uses several specialized representations for reactive behaviors, all of which support the same interface. See Chapter 12 for details.

Actions Lula's playback engine can handle arbitrary kinds of actions. Each action handles two basic operations, `prepare` and `terminate`. `Prepare` prepares for starting the action and establishes initial communication with the device that

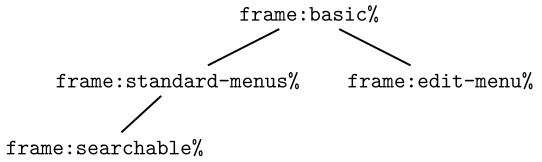

Figure 8.1: Frame classes in GUI framework.

executes it; `prepare` returns two procedures which start and stop the action, respectively. `Terminate` terminates the connection to the device. Section 9.5 has more details.

The set of action types is easily extensible through data-directed programming. (In fact, it is even extensible at runtime by virtue of DrScheme's dynamic module system.)

Script editor subwindows Lula's script editor consists of a number of nested editor widgets, each of which supports a common set of operations (implemented by a small number of mixins, as shown below in Section 8.7).

DrScheme's object system provides a convenient syntactic mechanism for creating representations with data-directed operations.

8.7 Parametric Inheritance

Traditional object systems offer *classes* as the primitive means of organization: A class is essentially a template for object creation; it specifies a set of attributes that each object created from the class has. Construction of a class happens by inheriting from an existing class and then adding and overriding selected attributes. In this arrangement, the original class is the *superclass*, the new class which inherits from the superclass is called the *subclass*.

Class construction via subclassing is usually a monolithic process: it names the superclass as well as the attributes it adds and overrides with one single syntactic construct. However, this mechanism is often not sufficiently flexible to address the needs of real programs.

Lula, for example, contains numerous classes for toplevel frames of its graphical user interface. These classes inherit from a variety of different classes from DrScheme's GUI library, depending on the features they need: some of these frames contain editors and must display the editor status, some contain help browsers with the appropriate menus, some frames are just plain. Depending on the function the frame has, there are various features it may need to offer as part of the Lula application:

- showing the Lula logo as an icon
- showing a Lula logo which changes color according to the source it controls
- be able to save the window geometry and restore it upon the next program run

Figure 8.1 shows the hierarchy of the relevant classes in DrScheme's GUI library. Lula now requires frame classes each of which inherits from one the classes in that hierarchy and includes a selection of the features from the above list. Figure 8.2 shows the desired hierarchy for frame classes capable of showing the Lula logo. Simple inheritance is not able to express this situation naturally; it requires the

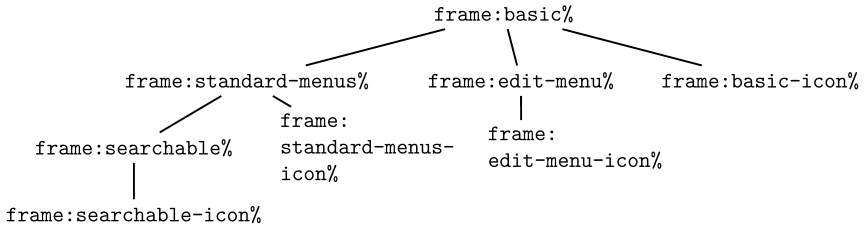

Figure 8.2: Frame classes with icon feature.

programmer to define separate subclasses for each of the original frame classes, each of which contains a copy of the code which installs the icon. The fundamental problem is that simple inheritance does not offer a suitable abstraction for isolated features applicable to a range of classes. The difficulties grow as the program needs frame classes with combinations of the above features.

Fortunately, classes in DrScheme are first-class values. This makes it possible to write procedures over classes. Specifically, it is possible to express the extension of a frame class by one the above features as a procedure from a class to class. Such a procedure which represents a feature applicable to all classes with a common interface is called a *mixin* [Flatt *et al.*, 1998, Findler and Flatt, 1998]. Here is the mixin for adding the icon:

```
(define lula-frame-mixin
  (lambda (frame%)
    (class frame% args
      (inherit set-icon)
      (sequence
        (apply super-init args)
        (set-icon lula-icon-bitmap lula-icon-mask 'large)
        (set-icon lula-icon-small-bitmap lula-icon-small-mask 'small)))))
```

With this definition, `lula-frame-mixin` is a procedure mapping a class with interface `frame<%>` to a subclass with interface `lula-frame<%>`. In the body of the class, the first line `(apply super-init args)` performs superclass initialization. The next two lines set the icon. Now, a program can simply use

```
(lula-frame-mixin a-frame%)
```

to create a subclass of `a-frame%` with the logo icon feature. Hence, mixins are a kind of *parametric inheritance*. Whereas this example is trivial, the mixins for other features on the list are considerably more involved. Mixins appear in a number of contexts in Lula, all associated with the construction of graphical user interfaces:

- DrScheme's application GUI framework [Flatt and Findler, 2000a] is completely organized as a collection of mixins, each providing a single feature to be added to a single kind of GUI widget class.
- Lula's GUI extension collection provides a number of GUI widget features as mixins. Among them are a mixin for specializing an interactive editor to accepting numeric input, a feature set for editors that can search for contents according to specified properties, and an extension to add auto-completion to an editor.
- A number of mixins allow connecting reactive values (as described in Chapter 12) to GUI components. This includes turning a button into an event, turning a slider into a behavior, or a behavior into a gauge. The library contains

only two mixins—`reactive-controller-mixin` and `reactive-view-mixin`—which connect GUI controllers and views with appropriate reactive values. The definitions of reactive sliders, gauges, buttons, check boxes, choice widgets etc. all result from applications of these two mixins.

- Lula's script editor consists of a number of nested editor widgets, each of which must communicate with the editor around it in the same way. The relevant functionality is a mixin.

8.8 Concurrent Programming

Lula is a naturally concurrent application: Lula needs to interact with the user while at the same time driving light changes as well as communicating with the hardware. Therefore, Lula runs as a collection of separately running *threads*[3]. Concurrent programming with threads requires support from both the programming language and its implementation. With Lula, the salient properties of DrScheme's thread system are the following:

Preemption DrScheme's thread system is *preemptive*—its scheduler transfers control between threads without their consent. This differs from non-preemptive (or *cooperative*) implementations of threads which require a thread to perform a system call before any transfer of control can take place. This is important in Lula, as several of its threads (the sampling thread, for example) exclusively perform computations.

Transparent I/O I/O blocking must not get in the way of thread scheduling. In Lula, the program must not stop when an interface device is not ready for input.

Asynchronous signals DrScheme supports *breaks* to send asynchronous signals between threads which are delivered as special exceptions. Lula uses breaks to implement disjunctive operations on threads, particularly to implement the merge of reactive behaviors.

Lula could actually benefit significantly from higher-level thread communication primitives as in CML [Reppy, 1988, Reppy, 1999]. Presently, DrScheme supports only semaphores and breaks [Flatt, 2000]. This would obviate the need for breaks and simplify much of the thread-related code. Currently, DrScheme's thread system has too much overhead to allow an implementation of CML.

[3]Note that here, threads serve as programming paradigm rather than means for improving performance by using multiple processors

Chapter 9

Application Structure

But
the way your head works is God's own
private mystery. What was it you
was thinkin'?
—Sailor in *Wild at Heart*

Whereas the previous chapter looked upon the Lula implementation from the standpoint of the programming language and the paradigms it provides, this chapter takes the opposite view: it presents the structure of the Lula system, and explains some of the abstractions that were instrumental in putting it together. The focus is on structural aspects: Lula, being a highly interactive piece of software, is structured as a *reactive network*. Hence, the techniques used to connect the pieces of the network form a kind of structural glue—they are the primary shaping force behind the Lula code.

The chapter begins by giving an overview of the subsystem structure of the Lula code. It goes on to explain the general nature of reactive networks. Lula contains a small number of libraries that provide the infrastructure needed for implementing reactive networks. The subsequent sections highlight two important subsystems—cues and actions—and how they connect to the reactive structure of the system.

9.1 A Bird's Eye View of Lula

Figure 9.1 shows a diagram of the most important subsystems of the Lula application. In the diagram, an arrow means "provides information for." The rectangles delimit subsystem collections with strong cohesion.

At the heart of the picture is the subsystem collection responsible for cues. Cues serve as the basis of basically all programmed lighting: the manual fader allows fading in cues, light fades specify a target cue, and reactive effects also use cues as their basic components for assembly.

On the left, the diagram shows the subsystems associated with the script editor and playback. The script editor allows pasting events into the script; events ultimately consist of *actions*. There are numerous action types; the script editor leaves all specific functionality to their implementations. More on that subject below in Section 9.5.

Once the user decides to play back a show, the *playback controller* takes over. It controls the show playback panel and creates subpanels associated with the various playback masters—the sequencers and parallelizers that start and stop the jobs of the actions. Again, the playback masters leave all implementation details specific to the action types to their implementations.

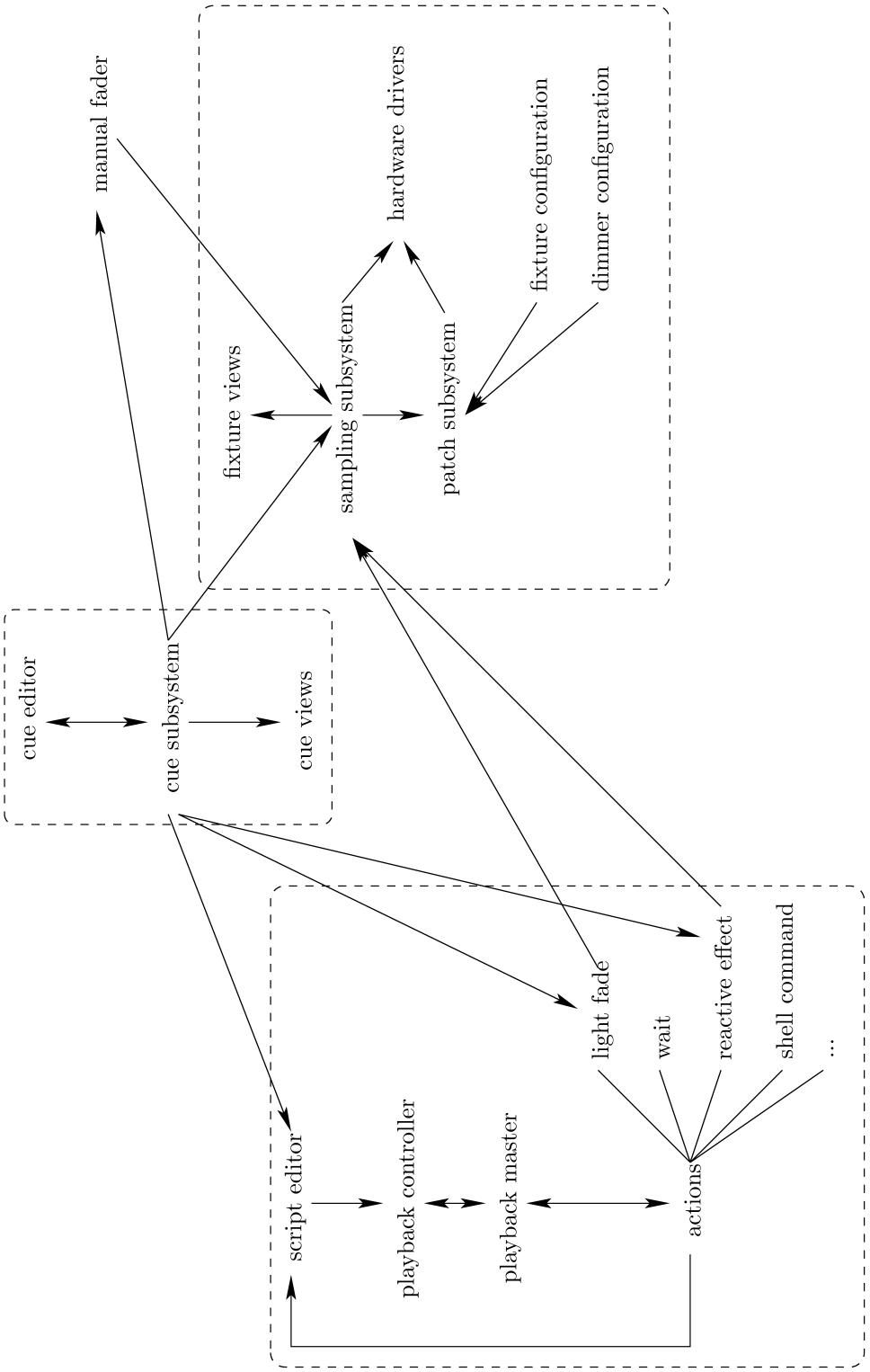

Figure 9.1: A bird's eye view of Lula.

The third collection of subsystems on the right concerns concerns the interface to the outside world. The sampling subsystem computes, from the various sources of lighting, unified parameter settings for all the fixtures in the system.

The sampling subsystem is also responsible for respecting dimmer-specific and fixture-specific dimmer curves, based on the information it gets from the configuration subsystems. Moreover, the sampling subsystem converts the fixture parameter settings into protocol-specific channel data. The hardware drivers retrieve that data and feed it into the hardware interfaces.

The manual fader takes up a somewhat lonely position: it uses the cue subsystem and provides data for the sampling subsystem.

9.2 Stratified Design

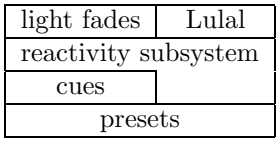

Figure 9.2: Stratified design in Lula.

Within the structure of Lula as a whole, the most important subsystems form a classic stratified design [Abelson *et al.*, 1996]: its basic data structure for representing fixture parameter settings is called *preset* (see Section 11.14; cues build upon it. The reactivity subsystem, the substrate for all animated light, builds upon cues and presets. Finally, both Lulal and light fade actions build upon the reactivity subsystem. Figure 9.2 shows the setup.

9.3 Reactive Networks

Lula is highly interactive software. This is not just in the sense that it has an interactive user interface. Rather, Lula keeps doing things even between user actions, and it must react to user requests to start new jobs, terminate old ones or modify their behavior. It is not just the user who provides impulses to change the behavior of the system: rather, the system produces such impulses internally as well. This makes Lula essentially a large *message network* [Peyton Jones *et al.*, 1999] or *reactive network*. The mechanisms which form the connections between the components are important for understanding the structure of Lula as a whole.

Figure 9.3 shows a segment of Lula's network structure: the cue, light fade, and manual fader subsystems are sources for lighting activities, encoded as some sort of *messages*. They supply data to the sampling subsystem which merges it into parameter settings. The parameter settings in turn flow into the patch subsystem which converts them into channel data for the hardware drivers. They also flow into the fixture view that displays the parameter settings to the user. Conceptually, the nodes at the top are *sources*, the oval nodes are *filters*, and the bottom nodes are *sinks* that turn information into observable effects. Flow networks like this one are common in interactive applications.

9.3.1 Push and Pull Architectures

An implementor of a reactive network needs address the question of control flow: how does a message actually travel between two nodes in a network? Which of the

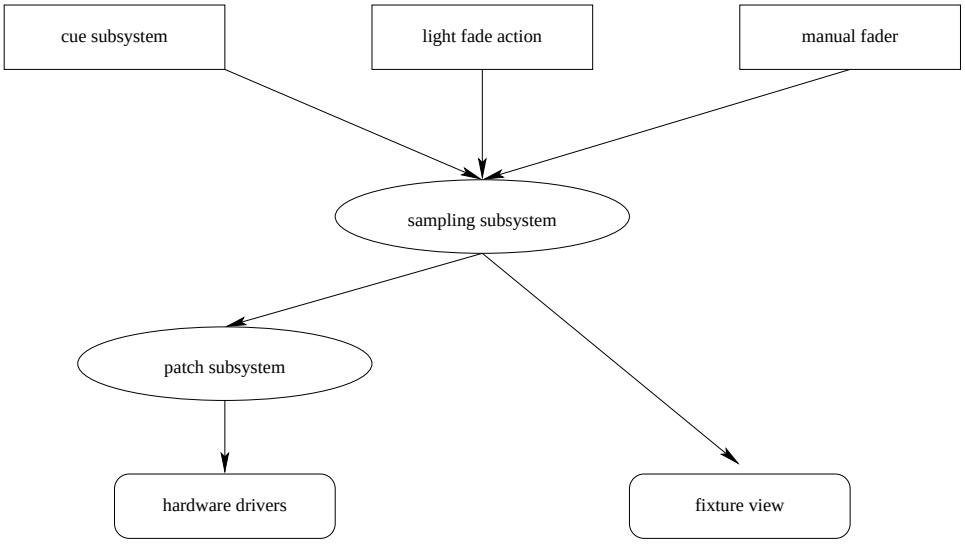

Figure 9.3: Reactive network in Lula.

two nodes takes the initiative, the originator or the recipient? There are two basic alternatives:

Push In a push-based setting, the originator, when it has a message to send, immediately "pushes" the message to the recipient, typically calling a procedure or method associated with it.

Pull In a pull-based architecture, the recipient decides when to receive a message: it "pulls" the message from the originator, possibly blocking if none is available.

The effect is, that in push architectures, availability drives the propagation of messages, whereas in pull architectures, it is demand that matters.

The composition of reactive networks is a natural part of the construction of graphical user interfaces. For example, the classic model-view-controller pattern [Krasner and Pope, 1988] prescribes an originator-recipient relationship between the model and the view as well as between the controller and the model. GUI toolkits typically have explicit support for constructing push-based or pull-based reactive networks. Most GUI toolkits based upon object-oriented programming support push. Examples include DrScheme's toolkit, MrEd [Flatt and Findler, 2000b], Smalltalk-80, and Java's Swing library [Walrath and Campione, 1999]. More recent designs, especially in the context of functional programming, favor pull. Examples are eXene for Standard ML [Gansner and Reppy, 1993] and Haggis for Haskell [Finne and Peyton Jones, 1995].

The push and pull architectures have different merits and disadvantages: push shines in situations where low latencies between message creation and message delivery are required. Pull architectures are more flexible because message recipients only need to pull messages when they are ready to process them. Pull is particularly effective for implementing Lula's reactivity subsystem—there, the temporal relationship between event occurrence and event handling can get quite complicated. Chapter 12 explains the implementation.

9.3.2 Implementing Push

The implementation of push architectures are usually based on the concept of the *callback*: A callback is a procedure or method registered with an originator of messages. The originator invokes the callback whenever a message is available, traditionally passing the message along with a reference to the originator itself.

In DrScheme, GUI controller components accept a procedure for a callback. Consider this example:

```
(define button (make-object button% "Click Me" parent
                            (lambda (self event)
                              (display "I've been clicked."))))
```

`Button` is a button with an associated callback that prints a short message; the event loop will invoke the callback whenever the user presses the button. In the above example, the callback is permanently associated with the controller—there is only one immediate recipient for button-click messages. In callback-based systems, an originator can continue processing messages only after the callback has returned. This means callbacks must not block or run for a long time. If a message is supposed to trigger a longer-running or blocking action, its callback needs to delegate to another thread. This raises potential synchronization issues.

When the programmer needs to hook up several recipients to a single originator, it is necessary to demultiplex the recipients off that callback. A convenient means to allow the dynamic adding and removal of recipients is the *observer pattern* [Gamma *et al.*, 1995]. It involves establishing a registry for recipient callbacks with the originator.

Note that with push, the originator keeps the recipient live. Even if the recipient stops having an effect (say, if its display window is deleted), the originator will still keep sending messages to it, as well as keeping the garbage collector from reclaiming the storage it occupies. This results in a *space leak* with potentially serious consequences. Preventing this unwanted effect requires a weak pointer mechanism [Peyton Jones *et al.*, 1999].

Lula naturally uses push to communicate with the GUI toolkit. Also, its playback controllers communicate with the playback masters via push. Moreover, Lula uses push to transfer sampled channel data to the hardware drivers—here, low latencies are important. In all of these contexts, the recipients and originators are fixed. Consequently, simple callback models suffice; it is not necessary to implement the observer pattern.

9.3.3 Implementing Pull

Pull is somewhat harder to implement than push. As an interactive application must be continually responsive to user actions, it is necessary to buffer messages between the time they become available and the time a recipient is ready to process it. Moreover, there is a natural concurrency of the originator code which delivers messages and the recipient code.

Lula uses *message queues* for buffering messages: a message queue is a thread-safe version of a standard queue data structure. It supports two operations: "send" which prepends a message to the queue, and "receive" which dequeues the last message. Thus, the message queue is a data structure with several potential originators and a single recipient of messages—once a recipient pulls out a message, it is gone. Consequently, message queues are sufficient for implementing the top part of Figure 9.3, but not for the bottom part.

In Lula, *exchanges* generalize over message queues: An exchange is a message buffer with potentially many originators as well as recipients. It contains a set of

message queues; one for each recipient. Sending a message to an exchange adds it to all of the queues. Consequently, an exchange provides insulation between an originator and its recipients, much like the observer pattern in the push model.

Just like the callback implementation of push, the message-queue implementation of pull requires some care to avoid space leaks: as messages arrive in a message queue, they take up space until the recipient retrieves them. Should the recipient die or be otherwise occupied, messages might pile up. For that reason, Lula's implementation of exchanges uses weak sets for holding the message queues; when the exchange is the only data structure holding on to a message queue, the garbage collector is able to reclaim it.

Lula uses exchanges for all communication involving the cue subsystem. Section 9.4 explains some details. As mentioned above, the reactivity subsystem is also pull-based.

9.3.4 Interfacing Push and Pull

As Lula uses both push and pull connections in its reactive network, it must be able to interface between the two models. Actually, message queues and exchanges already provide the necessary machinery to send a message from an originator which assumes push to a recipient which assumes pull. The following code fragment shows such a connection:

```
(define exchange (make-exchange))

(define button (make-object button% "Click Me" parent
                            (lambda (self event)
                              (exchange-send! exchange 'click))))

(define message-queue (exchange-make-message-queue exchange))
(define recipient-thread
  (thread
    (lambda ()
      (let loop ()
        (let ((message (message-queue-receive! message-queue)))
          ⟨process message⟩
          (loop))))))
```

The button—a push-based originator—sends a message to `exchange` on every click. The recipient, represented by a thread, adds a message queue to the exchange and loops pulling click messages out of the queue, processing them.

The other way around—connecting a pull-based originator with a push-based recipient—again requires a loop running in a thread:

```
(define originator-queue (exchange-make-message-queue exchange))
(define originator-thread
  (thread
    (lambda ()
      (let loop ()
        (let ((message (message-queue-receive! message-queue)))
          (callback message)
          (loop))))))
```

`Originator-thread` calls the `callback` with each message as it arrives. Note that, for this to work efficiently, blocking on `message-queue-receive!` must be cheap—polling is out of the question as the number of originators grows. In Lula's implementation of message queues, a semaphore counts the number of messages still

waiting in the queue, and thus blocking on an empty message queue costs nothing. The reactivity subsystem contains a somewhat more complicated mechanism called *placeholders* to implement efficient blocking. Once again, Chapter 12 has all the details.

9.4 The Cue Subsystem

The cue subsystem is at the heart of Lula; essentially all other system which function as sources of lighting parameter settings feed off it. The subsystem provides three kinds of information to dependent parts of Lula:

- the structural representation of each single cue, as specified by the user via the cue editor,
- the parameter settings specified by each single cue, computed from the structural information, and
- notification of cue creation and deletion.

Lula uses a term representation for the structure of the cue, the so-called *cue flat form*; the parameter settings are specified as *presets*—essentially sorted association lists. Chapter 11 has all the details on that.

This section focuses on the reactive connections of the cue subsystem with rest of Lula: it must reflect all changes made to the cues live—the user must be able to see the consequences of each change immediately. Now, a modification to one cue has an effect on the meaning of the cues which contain it, directly and indirectly. Lula must propagate these changes through the cue DAG. At the same time, this updating process must not block any other running subsystem. Lula uses the pull model for all communication emanating from the cue subsystem.

Preset Caching and Invalidation To avoid having to recompute the preset for a cue—the representation of its parameter settings—each cue caches the associated preset which is computed on-demand. Here is the definition for the cue record type using SRFI 9 notation [Kelsey, 1999]:

```
(define-record-type :cue
  (real-make-cue semaphore
                 name
                 flat-form
                 opt-preset upper-cues
                 structure-exchange preset-exchange)
  cue?
  (semaphore cue-semaphore)
  (name cue-name)
  (flat-form real-cue-flat-form unsafe-set-cue-flat-form!)
  (opt-preset real-cue-opt-preset unsafe-set-cue-opt-preset!)
  (upper-cues real-cue-upper-cues unsafe-set-cue-upper-cues!)
  (structure-exchange cue-structure-exchange)
  (preset-exchange cue-preset-exchange))
```

The semaphore in the `semaphore` field synchronizes writes to the `opt-preset` and `flat-form` fields:

The `opt-preset` field is the preset cache—it is `#f` if no preset has been computed yet and contains a preset otherwise. Each cue contains two exchanges, `structure-exchange` and `preset-exchange`. The cue subsystem sends a message

to `structure-exchange` whenever it changes the structure of the cue by replacing the cue's flat form in the `flat-form` field by another. The constructor for cues always adds a special global message queue `*bust-cue-cache-queue*` to the cue's preset exchange; a dedicated thread monitors this queue to propagate preset changes through the cue hierarchy asynchronously:

```
(define (make-cue name)
  (let ((cue
         (real-make-cue (make-semaphore 1)
                        name
                        (make-cue-rep (make-flat-form '()))
                        #f
                        '()
                        (make-exchange) (make-exchange))))
    (exchange-add-message-queue! (cue-preset-exchange cue)
                                 *bust-cue-cache-queue*)
    cue))
```

The cue subsystem sends a message to the `preset-exchange` whenever the cue's associated preset changes. This may happen because the cue's structure has changed, or because the parameter settings of any of the subcues have changed. Here is the procedure that signals a change to a cue's preset, invalidating the `opt-preset` field:

```
(define (cue-notify-preset-change! cue)
  (with-semaphore (cue-semaphore cue)
    (lambda ()
      (unsafe-set-cue-opt-preset! cue #f)))
  (exchange-send! (cue-preset-exchange cue) cue))
```

A structure change is always a preset change as well:

```
(define (cue-notify-structure-change! cue)
  (cue-notify-preset-change! cue)
  (exchange-send! (cue-structure-exchange cue) cue))
```

The procedure responsible for actually making a structure change to cue is `set-cue-flat-form!` The procedure must notify the cues of all subcues, called *under cues* in Lula. `Set-cue-flat-form!` does the job, with the help of `cue-flat-form-under-cues` which collects the under cues, and `swap-under-cues!` that adjusts the `upper-cues` components of the under cues. Finally, the `set-cue-flat-form!` procedure notifies the structure exchange.

```
(define (set-cue-flat-form! cue flat-form)
  (let* ((old-flat-form (real-cue-flat-form cue))
         (old-under-cues (cue-flat-form-under-cues old-flat-form))
         (new-under-cues (cue-flat-form-under-cues flat-form)))
    (with-semaphore (cue-semaphore cue)
      (lambda ()
        (unsafe-set-cue-flat-form! cue flat-form)))
    (swap-under-cues! cue old-under-cues new-under-cues))
  (cue-notify-structure-change! cue))
```

`Set-cue-flat-form!` only notifies the structure exchange of the cue whose flat form has changed. This change must propagate upwards through the hierarchy, as mentioned above. Therefore, the cue subsystem runs a dedicated preset invalidation thread which runs the `bust-cue-caches` procedure. `Bust-cue-caches` waits for messages on `*bust-cue-cache-queue*` and, upon receiving one, notifies the cue:

```
(define (bust-cue-caches)
  (let loop ()
    (let ((cue (message-queue-receive! *bust-cue-cache-queue*)))
      (with-semaphore (cue-semaphore cue)
        (lambda ()
          (for-each cue-notify-preset-change!
                    (real-cue-upper-cues cue))))
      (loop))))
```

Thus, preset recomputation happens concurrently with the rest of the system—cascading preset changes do not block the rest of the application.

Cue Creation and Deletion The cue subsystem also uses an exchange to register creations of new cues as well as cue deletions:

```
(define *cue-list-exchange* (make-exchange))
```

The `install-cue!` procedure (which may run asynchronously, and hence uses a semaphore to synchronize) adds a newly created cue to the system and notifies the exchange with an `install` message:

```
(define (install-cue! cue)
  (with-semaphore *cues-semaphore*
    (lambda ()
      (set! *cues* (insert-sorted cue *cues* cue<))
      (exchange-send! *cue-list-exchange* (cons 'install cue)))))
```

A recipient of these changes calls `make-cue-list-change-message-queue` to create a message queue that will receive these messages:

```
(define (make-cue-list-change-message-queue)
  (exchange-make-message-queue *cue-list-exchange*))
```

Cue Views A number of Lula's windows display a view of the cue hierarchy via a treelist GUI component (see Chapter 6). Such a cue view must monitor both cue structure changes as well as creations and deletions of cues. Hence, it provides a good example for a recipient of cue information.

Each cue view maintains a single message queue which receives both cue structure changes as well as cue creation and addition messages:

```
(private
 (message-queue (make-cue-list-change-message-queue)))
```

(`Private` is a clause in DrScheme's object system denoting a private instance variable.)

Furthermore, whenever the user "opens" a cue in a cue view to look at this structure, the cue view must monitor all changes to this structure. The `on-item-opened` method which the treelist component calls in this case adds `message-queue` to the cue's exchange in that case:

```
(override
 (on-item-opened
  (lambda (item)
    (open-compound-item item get-under-cues)
    (if (not (item-ever-opened? item))
        (exchange-add-message-queue!
          (cue-structure-exchange (get-item-cue item))
           message-queue)))))
```

Furthermore, each cue view has its own thread which monitors the message queue and updates the display:

```
(private
  (update-thread
   (thread
    (lambda ()
      (let loop ()
        (let ((message (message-queue-receive message-queue)))
          (if (cue? message)
              ;; cue structure change
              (for-each-observer (lambda (item)
                                   (update-item item get-under-cues))
                                 message)
              ;; cue creation or deletion
              (update-all-cues))
          (loop)))))))
```

9.5 Representing Actions

A critical aspect of the Lula code is its representation of actions as they are the most reactive part of the system. The choice of representation is critical for the implementation of playback masters and playback controllers which supervise the controlled execution of the jobs associated with the actions. Actions use the push model for communication with the rest of Lula.

The action representation must accommodate a number of different types of actions. Here are some examples:

- light fade
- light effect
- sound playback (a future extension)
- pause for a certain time
- wait indefinitely

To abstract over these different actions, it is necessary to divide the life cycle of an action execution into different phases.

Preparation An activity might need preparation. For example, if the action specifies playing a CD track, Lula needs to spin up the CD drive so that playback can start instantaneously.

Activity The "go" signal from the user triggers the activity of the action.

Relaxation After some time, the activity completes by itself or the user requests that it complete. A "stop" signal ends the activity. However, it may be necessary to keep a residue of the activity. For example, a completed light fade must keep the stage lit.

Nonexistence Finally, even the residue must go—either because a subsequent action takes the place of the current one, or because the show is over. A "terminate" signal condemns the action to nonexistence.

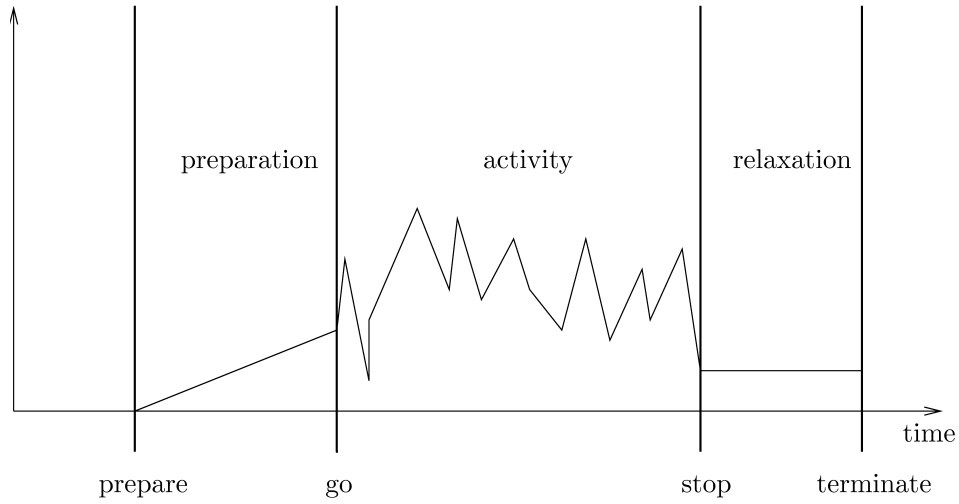

Figure 9.4: The life cycle of an action.

Figure 9.4 shows a plot of these phases.

Actions use a data-directed representation using DrScheme's object system. For action execution, its interface contains three basic methods:

```
(define action<%>
  (interface ()
    get-device
    prepare
    terminate))
```

`Get-device` identifies an abstract device associated with the action. This is important for action sequencing: for instance, a sound-playback action must not terminate a prior light fade. Therefore, a sequencer will keep the actions for different devices separate from each other.

`Prepare` starts the preparation phase of an action. It returns two thunks, `go` and `stop`. Calling `go` starts the activity, calling `stop` stops it and starts the relaxation phase.

`Prepare` takes as arguments a callback which is invoked upon the end of the activity—no matter if it was caused by the natural end of the activity or user intervention. The callback, when invoked, will receive a so-called *token* representing the residue of the action. When `prepare` is passed such a token which belongs to a prior action, it will initiate a transition between the prior action and the current one. For a light fade, for instance, this involves keeping the lights up.

Part V

The Look of Lula

Sailor, that ozone layer is
disappearin'. Seems to me the
government could do somethin' about
it. One of these mornings the
sun'll come up and burn a hole clean
through the planet like an X-Ray.
—Lula in *Wild at Heart*

One of the two principal tasks of the lighting designer is the construction of *looks*—a look is a configuration of the lighting fixtures that—together with the set and the actors—creates a stage picture.

From the viewpoint of the lighting operator, a look is a setting for the parameters of the fixtures belonging to a show. This is actually the way most commercial lighting consoles view looks, and the operator spends her time setting specific parameters of specific fixtures via the operation of the console keyboards as well as faders and fader-like controls.

However, there is more to lighting design than just arbitrary collections of fixtures and their parameter settings. In fact, designed light has structure which emerges first as the ideas of the director and the lighting designer evolve, later as the lighting operator puts them into practice. Notably, good directors and designers will specify their ideas at a level that is independent of the concrete stage environment and the number, nature and location of the available fixtures. This is important for touring productions which must function in wildly varying stage environments but where the desired structure of the lighting remains the same.

Unfortunately, the means of commercially available consoles for dealing with structure in looks is, in most cases and at worst, non-existent, at best awkward. Programming the lighting for a show at the level of single fixtures and their parameters is a lengthy, error-prone and inflexible process. Changes to such installations "after the fact" are hard and time-consuming. Presently, the market seems to feature only two hierarchical consoles which are, in principle, able to model look structure. However, the hierarchical nature of these consoles is hidden away in the documentation and awkward to use.

Lula's approach to look design is radically different:

- Lula's model of a look is *structural*. Programming progresses along the conceptual structure of the design, as do all later changes.
- Lula allows *componential lighting design* which views a look as a combination of components, each of which may itself consist of components, and so on. The smallest components of a look are individual fixtures.
- Componential lighting design allows, to a high degree, *venue-independent programming*. Most programming does not happen at the level of individual fixtures but rather that of higher-level structures in the design. The actual fixtures of a given installation are interchangeable, and only require the operator to change those aspects of the programming which relate to the changes in the environment. The operator can preserve and re-use the conceptual components.

This part of the dissertation describes Lula's model for looks and lighting components in Chapter 10 from the viewpoint of the lighting designer. Chapter 11 gives a solid algebraic foundation for the model which has proven crucial for designing the user interface and verifying the soundness of the model.

Chapter 10

Requirements for Look Construction Systems

Johnnie, you take a good look at me,
baby, cause you gonna haf'ta watch
close to know when we do it to ya ...
—Juana in *Wild at Heart*

In lighting design, the design of *looks* is the creation on configurations of light intensities and other parameter setting to create a stage picture. Few looks consist only of one picture—even lighting a single small square on stage usually requires more than one light source (see Chapter 4). Even on small stages, a look might be a complicated composition of many fixtures.

As the lighting designer and operator create a look, they impose organization upon the fixtures. Their ability to achieve the stage picture they want depends on the ability of their lighting console to represent this organization. Typically, this means dividing a complicated look into several, more or less independent parts, and composing them to create the picture. As the preparation of the show progresses, other factors impact the lighting work: Does a console allow making changes to a look programmed earlier? How difficult is this? How robust and flexible is the programming in the face of these changes?

This chapter examines the issues involved in look construction from the viewpoint of the lighting designer and provides motivation for Lula's specific look construction system which is based on the notion of *cues*. It starts with an account of the basic terminology and a short review of the methods of commercial lighting consoles for dealing with look construction, reiterating some material from Chapter 5.

10.1 Componential Lighting Design

As opposed to most traditional lighting consoles, Lula actually manages and stores structural information about a look, rather than just viewing it as a collection of fixture/parameter/value triples. Moreover, Lula allows the modification of the current show on a structural level. This provides significant added flexibility over traditional models.

The central idea underlying Lula's model for look construction is *componential lighting design*. Lula's basic assumption is that lighting design and programming evolves in components. A director or lighting designer will see a show as a sequence of looks or pictures, each of which requires its lighting to correspond to structural

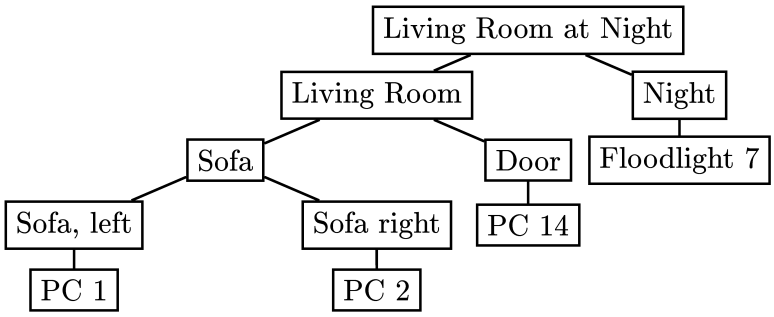

Figure 10.1: A look with hierarchical structure.

aspects of the show. As for the look aspects of lighting, examples include the following—all taken from theater (cf. Chapter 4):

- "In Scene 4, Bert is sitting on that armchair. We need light on him."
- "We need the dining table, and good facial light on the people sitting downstage, left, and right."
- "Scene 3 plays at night. We want the whole stage lit, but dimly, and something resembling moonlight."
- "I want the whole stage lit except for the downstage-left corner."
- "The square in the middle of the stage is the principal acting arena. We need focus on the square and less emphasis on the surrounding area."

The first example asks for light on a specific actor, maybe the smallest unit of lighting in classic theater lighting. For good facial lighting, several fixtures are necessary (see Chapter 4). However, the conceptual component is the lighting on the armchair, not the specific combination of fixtures and intensities used to implement it.

The second example already provides more structure than the first one: three actors are sitting at the dining table, and their faces need to be visible. It may be possible to light two of them with a common fixture, but generally, the lighting component "three people sitting at the dining table" includes three subcomponents for the three actors. (Additional lighting might be necessary for the table itself, or other areas around the table, creating additional subcomponents.) So far, the components are playing areas or combinations of them (see Section 4.5.1).

The third example also provides structure: dim lighting on the stage and moonlight, possibly provided by a dim blue-colored floodlight. Presumably, the lighting on the stage also consists of several subcomponents. However, at the level of description of the moonlit stage, this is not important.

Whereas the first three examples are all *additive* in a sense—combining subcomponents means adding them visually on stage. The fourth example is *subtractive*: in the resulting look, a corner is missing from the "stage" component.

The structure of the final example is less clear: it might be additive, consisting of subcomponent for the central square and the surrounding stage. It could also consist of a subcomponent for the entire stage with the area around the stage forced to lower intensities.

The examination even of designs for small shows and small stages often reveals a surprisingly rich hierarchical structure. Part of this structure often already exists in the director's script; the lighting designer provides the rest. Figure 10.1 shows

a structural view onto a component "Living Room at Night": it has two subcomponents "Living Room" and "Night", both of which have subcomponents of their own. A single flood provides the "Night Lighting" which provides the implementation of the conceptual "Night" entity. "Living Room," on the other hand, has two conceptual subcomponents, each in turn with their own structure. The entire structure forms a tree. All the components in a show form a hierarchy, represented by a directed acyclic graph.

Traditional lighting consoles offer grouping facilities which allow some degree of component assembly, usually called *presets*, *masters*, or *submasters*. However, these consoles usually limit the depth of the hierarchy to two or three, and components exist *during construction only*: the structure is visible in the configuration of faders on the operating board. As soon as the operator presses the "Record" button, the console forgets the structural aspects of the current console, saving only fixture/parameter/value settings.

The lack of structural information in the console's memory means that changes to an already-recorded cue happen only at the fixture/parameter/value level, rather than on the cue structure, as they should. This frequently leads massive typing efforts necessary to apply a simple change to a structural aspect. Consider a show with a large number of light changes; a single fixture breaks, and has to be replaced it with a different, brighter fixture (or, worse, several smaller fixtures). The operator now needs to adjust the intensity channel of the fixture for every single cue which is part of the show. Tracking consoles and consoles separating between presets and cues can sometimes alleviate the problem with careful programming, but cannot solve it in general.

10.2 Cues

Lula's model for looks builds on the *cue*; a cue is a component of a look[1]. The "smallest" cues are single fixtures, the "largest" cues are complete looks. Lula allows combining "smaller" cues into "bigger" ones. The cue subsystem is the central innovation in Lula. It has a number of desirable properties which existing consoles do not have or possess only to a limited degree:

- The model allows for *componential lighting design* which allows the designer to construct a library of building blocks which may be combined to form new cues.
- The model allows for *venue-independent programming*: a designer can separate the conceptual aspects of a design ("What do I want to see on stage?") from the physical ones ("What fixtures have to perform what function to make it happen?") This allows the offline preparation of major parts of a design. Moreover, it allows touring shows to preserve most of their programming. Offline preparation is a crucial feature, as onstage time is often expensive and limited.
- Componential lighting design leads to programming which is both flexible and robust: Lula allows changes at the structural level at any time during the lighting design (even during a running show). Robustness is a consequence of venue independence: as the physical environment changes, much of the programming of a show can survive.

Lula's model is based on algebraic principles; Chapter 11 gives a detailed accounts of its foundations and its properties. The algebraic design enforces regularity and

[1] In retrospect, the choice of terminology seems unfortunate, as in theater language, a „cue" denotes a marker for a point in time.

usability of the model. Fortunately, the user does not have to deal with algebra in order to use Lula; she sees only the pleasant consequences.

In the terminology of this dissertation, a cue stands for a collection of fixture parameter settings. A cue is a component of a look. Actually, a look is itself a cue—this dissertation will use the term *look cue* in contexts where there is a difference between them and "ordinary cues."

When a fixture is part of the lighting specified by a cue, the cue *contains* or *specifies* the fixture. More precisely, a cue specifies settings for the parameters of the fixtures it contains.

10.3 Compositionality

The key to designing an effective model for cues lies in preserving a property called *compositionality*. Compositionality—a term originating in denotational semantics—essentially means that the meaning of a composite object is completely determined by the meaning (rather than the structure) of its parts. Applied to lighting, compositionality ensures that a cue hierarchy remains manageable, and that making changes to the cues will actually have the effects an operator naturally expects. Commercial consoles typically do not implement compositionality, but rather use low-level ad-hoc mechanisms to control the meaning of composite lighting. This section explores the cause for this and present Lula's approach to the problem.

The key benefit in representing a lighting component by its name is *abstraction*: the cue name should abstract from the details of its "implementation"—the concrete fixtures and their parameter settings that create its stage picture. Rather, what counts is the look the lighting component creates. In principle, the implementation is replaceable.

HTP and Compositionality When an operator creates a composite cue from two subcues, it is easy to describe the outcome if the implementation of the subcues do not share any fixtures. As the composite cue is invoked, all the fixtures involved in the first subcue will be at the intensities and positions it specifies; likewise for the second subcue. But what if there is a conflict—a fixture is part of the implementations of both the first and second subcue? For traditional theatrical fixtures which only have an intensity control, the solution is simple: the composite cue will drive the fixture intensity at the maximum of the intensities specified in the subcues. This means the composite cue will illuminate everything on stage at least as brightly as each of its subcues.

In lighting design, this principle—a maximum or lowest upper bound in Mathematics terminology—*highest takes precedence* or *HTP*, is simple and effective. In fact, HTP by itself is sufficient for most theatrical applications; up until and including Lula 3, this was the only strategy for cue combination Lula supported.

Unfortunately, HTP is sometimes undesirable and sometimes not applicable. Consider moving lights: a composite cue might consist of two subcues, each of which include a moving light at different positions. Usually, moving lights accept their position in two numbers, one for pan, the other for tilt. Pan/tilt has no natural notion of "maximum." Moreover, "maximum" as applied to pan/tilt does not correspond to any intuitive counterpart in the lighting. Specifically, it would depend on the orientation of the actual lights on the ceiling. For example, if the composition operator were to use element-wise maximum as for intensities, the result would become completely unpredictable, sometimes using pan from one subcue, and tilt from another. Hence, two subcues specifying different positions for a common moving light are fundamentally incompatible as far as HTP combination is concerned.

Combining Non-Intensity Parameters As it turns out, HTP is not applicable to most non-intensity parameters of modern multi-function fixtures: what is the maximum of two colors? Two beam settings? Two gobos? Therefore, a lighting console must offer a way to combine subcues which allows overlap, yet resolves possible conflicts. This makes it necessary to give one subcue or the other precedence. Most lighting consoles use *latest takes precedence* (*LTP*). LTP is a property of a single output parameter or slot. Should two subcues to be combined share a fixture with an LTP parameter, the console will choose the value specified by the latter of the two subcues. (See Chapter 5 for more details on how commercial consoles deal with conflict resolution.)

The LTP strategy reliably resolves conflict, but destroys compositionality: LTP is a property which does not correspond to any visible aspect of the lighting. Therefore, it is impossible to predict the outcome of a combination of the two subcues looking at the meaning of (the light generated by) the two subcues separately. Rather, determining the meaning of the composite cue require knowing a non-visible, implicit aspect of their implementation. Thus, LTP conceptually happens at the implementation level, not at the structural level. This leads to inflexible, hard-to-modify programming. Consequently, Lula does not support LTP for conflict resolution.

Lula chooses a different, explicit approach to conflict avoidance: Lula offers a novel combination operator called *restriction*. The restriction of some subcue A with another subcue B will look like A in places where A does not overlap with B, and like B in all others. Essentially, B takes precedence over A.

Lula's approach to conflict resolution forces the operator to be explicit about situations which she would normally handle by trial-and-error on a console which supports LTP. However, Lula rewards the explicitness introduced by the use of restrictions with more flexibility and robustness when it comes to using the resulting cues in different contexts and modifying them later.

10.4 Cue Combination

This section summarizes the different means of cue combination Lula offers the operator for constructing cues from subcues. Besides HTP and restriction, there is a third one called *difference*.

Besides satisfying most practical needs, this set of "cue combinators" also has pleasant algebraic properties which in turn help structure the user interface in an intuitive way. Chapter 11 has the details, along with a formal specification of cue semantics. This section also suggests a notation for subcue combination.

HTP For two subcues a and b, $a \sqcup b$ is their *HTP*. $a \sqcup b$ specifies all fixtures of both a and b at the same positions, colors, beams etc. as they appear in the respective subcues. $a \sqcup b$ will specify the intensity of each fixture appearing in both a and b at intensities i_1 and i_2 as $\max(i_1, i_2)$.

The HTP only exists if s_1 and s_2 are *compatible*. Intuitively, compatibility means it is actually possible to achieve the visual effect of s_1 and s_2 simultaneously as far as visibility is concerned. Specifically, two subcues are compatible if they do not both specify a non-intensity parameter for a common fixture.

Restriction For two subcues a and b, $a /\!\!/ b$ is the *restriction* of a *with* b. Restriction applies to any two subcues. The restriction specifies all fixtures in either a or b. Fixtures only a specifies appear in $a /\!\!/ b$ as they appear in a. All others appear as they do in b. Thus, $a /\!\!/ b$ is a combination of a and b which assigns precedence to b.

Difference Lula's cue model includes a third means of combination. Whereas HTP and restriction always "add" the set of fixtures specified in subcues, subtraction reduces it. For two subcues a and b, $a \setminus b$ is the *difference* between a and b. The difference includes all fixtures appearing in a but *not* appearing in b at the same parameter values as in a.

In conjunction with differences, it is also sometimes useful to subtract the *complement* of a cue rather than the cue itself. $a \setminus \overline{b}$ is the difference between a and the complement of b. The complement of a cue contains all fixtures *not* in the cue. Thus, subtracting the complement of a cue is somewhat like placing a stencil over a cue: $a \setminus \overline{b}$ contains all fixtures that a and b have in common, at the parameter setting a specifies. Complements appear only in conjunction with differences, and indeed only make sense in that context.

Fixtures at Zero Intensity Restriction of one subcue with another raises another issue of cue semantics affecting compositionality: what is the meaning of a cue which includes fixtures at zero intensity? Restriction and difference reveal a difference between a cue B including a fixture f at zero intensity and another cue B' identical to B in look, but which does not include the fixture in the first place: Restricting a cue A which specifies fixture f at non-zero intensity with B will result in a new cue with f off as well. On the other hand, restriction with B' will leave f on as specified with A.

The difference between a cue specifying a fixture at zero intensity and a cue which does not contain the fixture at all is not visible, at least not on stage. (Lula's user interface does show the difference.) It only becomes apparent in combination with other cues. This breaks compositionality: a non-visible aspect of a cue has impact on its behavior. On the other hand, it is a situation with no immediately obvious application, so a reasonable approach to the problem would be to forbid zero-intensity fixtures as parts of cues.

10.5 Examples Review

Here is how the cue combinators apply to the examples from Section 10.1:

- "In Scene 4, Bert is sitting on that armchair. We need light on him."

  This instruction does not contain any explicit structure.

- "We need the dining table, and good facial light on the people sitting downstage, left, and right."

  This instruction is a typical application of the HTP principle to subcues for the dining table, and the facial lights on the downstage, stageleft, and stageright areas.

- "Scene 3 plays at night. We want the whole stage lit, but dimly, and something resembling moonlight."

  Again, a typical application of HTP to two components.

- "I want the whole stage lit except for the downstage-left corner."

  This is an application of subtraction: presumably, there is a cue for the entire stage as well as one for the downstage-left. The cue requested here is the difference between one and the other.

- "The square in the middle of the stage is the principal acting arena. We need focus on the square and less emphasis on the surrounding area."

This is a possible application for restriction: if there are cues for the entire stage and the central square respectively, this could be a cue with the entire stage restricted by a dimmed-down version of the central square lighting.

10.6 Cue Transformation

It is rarely sufficient that compound cues arise simply by application of the three cue combinators described in the previous section. Typically, a subcue needed to compose a compound cue is not just another cue, but will need some change applied to it before it works in the compound cue.

The most common such change to a cue for inclusion as a subcue is the application of a relative intensity. Typically, the lighting designer applies such an intensity change so that the subcue will look good together with other subcues that are part of the compound cue. Such a relative intensity change is purely an adjustment to account for the lack of "absolute sight" on the part of the designer. However, such an intensity change might also be a director's instruction. Consider the examples in Section 10.1 design calling for softer overall lighting or just for parts of a cue to be darker than usual.

The idea of a uniform change to a cue extends to other parameters beside intensity. Here are some examples of such changes:

- "I want this cue, but in green."
- "I want this cue, but without any red."
- "I want this cue [composed of moving lights], but shifted two feet to the left."
- "I want this cue, but with softer beams."

These are all examples of *transformed* versions of cues, and the selection of transformations available has a strong impact on the expressiveness of the cue language. Here are some of the transformations Lula currently supports. The set is easily extensible:

Intensity Scale An intensity-scale transformation scales the intensity of the fixtures in a cue uniformly by some factor. Ideally, the application of a 0.5 intensity-scale transformation to a cue will yield a cue "half as bright."

Color Set A color-set transformation sets the color of all fixtures in a cue that allow color control to a certain color. Depending on the kind of fixture, the actual color generated might be approximate.

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

X/Y/Z Set An X/Y/Z-set transformation specifies stage coordinates for the moving lights of the resulting cues to focus on. (As moving lights usually operate in polar coordinates, the operator needs to perform calibration on all moving lights for this to work.)

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.

Hence, in Lula, a subcue is a cue together with a set of transformations. In the construction of compound cues, the cue combinators apply to subcues in this sense rather than untransformed "ordinary" cues.

Note that, in a cue $a \setminus b$, transformation of b has no effect on the difference: all fixtures that b contains are not part of the compound cue. A transformation has no effect on the set of fixtures a cue contains.

Chapter 11

An Algebra of Cues

Yeah . . . yeah . . . I guess so . . .
That kind'a money'd get us a long
way down that yellow brick road . . .

—Sailor in *Wild at Heart*

This chapter gives a formal description of Lula's cue model. The model is based on a small term language with two central sorts—one for cues, and one for uniform parameter transformations of cues. The semantics of the language yields fixture parameter settings directly.

The cue language is effectively a *little language* [Bentley, 1986] or *domain-specific language* (DSL) [van Deursen *et al.*, 2000], and has strong methodological similarities to Haskore [Hudak *et al.*, 1996], a language for describing sheet music. The cue language, itself having two levels is the basis for the animation layer, described in Part VI of this dissertation, itself a DSL, and thus contributes to the stratified design at the core of Lula.

The term structure of the language is not visible to the Lula user. Rather, the user creates and edits cue terms via a graphical user interface. However, it would be perfectly feasible to present an alternative view and control onto cues, employing traditional term notation and a text or structural editor.

The work in this chapter was instrumental in designing Lula's graphical user interface for cues. The main problem which needed solving is that graphical user interfaces are conceptually flat. The display and control of tree-like data is possible through the use of box-and-pointer diagrams, but is rather unwieldy for the user as compare with more traditional user-interface elements such as list-boxes and tab controls. Fortunately, all cue terms in Lula are equivalent to a term with depth 3; this chapter contains a proof of this property. The resulting representation, *cue flat form*, is perfectly amenable to standard GUI techniques.

Bringing formal methods into a domain as practically oriented as lighting may seem excessive. However, the previous chapter has shown that semantic questions do arise in day-to-day lighting, and the resolution of the resulting ambiguities is a pertinent issue in current console design and operation.

Applying semantics to lighting yields some interesting parallels to the denotational semantics of programming languages: if the primary purpose of light on stage is to make things visible and thereby convey information, cues exhibit a semantic structure similar to semantic domains.

11.1 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 11.9 shows how.

Cues form an *algebraic specification* [Wirsing, 1990, Klaeren, 1983]. The basic signature for cues builds on a two primitive sorts:

- *factor* represents a scalar factor; the scale function uniformly scales the intensity of a cue.
- *fixture* represents a fixture, with an assumed constant for every fixture.

signature $\Sigma_{\text{CUE0}} \equiv$
  sort *factor*
  sort *fixture*
  sort *cue*
  functions
    fromfixture : *fixture* → *cue*
    scale : *factor*, *cue* → *cue*
    black :→ *cue*
    $\_\sqcup\_$: *cue*, *cue* → *cue*
    $\_/\!/\_$: *cue*, *cue* → *cue*
    $\_ \setminus \_$: *cue*, *cue* → *cue*
endsignature

Figure 11.1: Signature for simple cues.

Figure 11.1 shows the signature of the algebraic specification. 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 (see Section 10.4).

In the language of the specification, the cue in Figure 10.1 in Section 10.1 has the following definition, disregarding intensity adjustments:

$$\begin{aligned}
\boxed{\texttt{Sofa, left}} &:= \text{fromfixture}(\texttt{PC 1}) \\
\boxed{\texttt{Sofa, right}} &:= \text{fromfixture}(\texttt{PC 2}) \\
\boxed{\texttt{Sofa}} &:= \boxed{\texttt{Sofa, left}} \sqcup \boxed{\texttt{Sofa, right}} \\
\boxed{\texttt{Door}} &:= \text{fromfixture}(\texttt{PC 14}) \\
\boxed{\texttt{Living Room}} &:= \boxed{\texttt{Sofa}} \sqcup \boxed{\texttt{Door}} \\
\boxed{\texttt{Night}} &:= \text{fromfixture}(\texttt{Floodlight 7}) \\
\boxed{\texttt{Living Room at Night}} &:= \boxed{\texttt{Living Room}} \sqcup \boxed{\texttt{Night}}
\end{aligned}$$

Should, say, the sofa need less intensity for the right side to be properly balanced, the corresponding definition could change this way:

$$\boxed{\texttt{Sofa}} := \boxed{\texttt{Sofa, left}} \sqcup \text{scale}(0.7, \boxed{\texttt{Sofa, right}})$$

11.2 Carrier Sets for Cues

The normal procedure for constructing an algebraic specification is to write the specification first and worry about carrier sets and semantics later. In this case, however, the actual carrier already live in the real world. Thus, it makes sense to start with an attempt to represent reality semantically, and backtrack from there to an equational specification.

A meaning for Σ_{CUE0} is a Σ_{CUE0}-algebra. Such an algebra for Σ_{CUE0} consists of carrier sets or domains for *factor*, *fixture*, and *cue*, as well as meanings for the functions.

A Σ_{CUE0}-algebra A^0 represents a natural intuition in the realm of theatrical lighting and centers around the notion of the cue. A cue conceptually has the following components:

- a set of fixtures that are part of the cue, and
- intensities for these fixtures.

The notion of "cue contains fixture" is explicit. Thus, 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.

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

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

The natural Σ_{CUE0}-algebra is called A^0. Its carrier set A^0_{fixture} for *fixture* contains elements for all fixtures. A^0_{cue} is a set with:

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

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

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

Factors are non-negative real numbers:

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

11.3 Semantics of Cues

The next step in constructing A^0 is assigning meaning to its constants, black, scale, fromfixture, apply as well as for HTP, restriction, and subtraction.

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 cue in the exponent:

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

The difference combinator is the set-theoretic difference between the fixtures contained in the operand cue. Note that the difference does not consult the intensity assignment of the second operand.

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

11.4 Axioms and Theorems for Cues

The A^0 algebra is a good starting point for deriving fundamental algebraic properties of cues to develop Σ_{CUE0} into an algebraic specification. It has a number of pleasant properties which will form an axiomatic basis for the specification. Here is the most immediate one[1]:

11.1 Axiom
HTP is commutative and associative.

Proof *Follows from the commutativity and associativity of* $\cup$ *and* max. □

HTP and restriction are trivially idempotent:

11.2 Axiom
HTP and restriction are idempotent. For every cue c:

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

Proof □

A number of trivial axioms concern the behavior of the black cue:

11.3 Axiom
For any cue c:

$$c \sqcup^{A^0} \text{black}^{A^0} = c \tag{11.1}$$
$$c /\!\!/^{A^0} \text{black}^{A^0} = c \tag{11.2}$$
$$\text{black} /\!\!/^{A^0} c = c \tag{11.3}$$
$$c \setminus^{A^0} \text{black}^{A^0} = c \tag{11.4}$$
$$\text{black}^{A^0} \setminus^{A^0} c = \text{black}^{A^0} \tag{11.5}$$
$$c \setminus^{A^0} c = \text{black}^{A^0} \tag{11.6}$$

Proof □

[1] These properties 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 next insight is that restriction is expressible in terms of HTP and difference:

11.4 Lemma
In A^0, for any cues a and b:

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

Proof *Let $(F_1, p_1) := a$ and $(F_2, p_2) := b$. Furthermore, let*

$$(F, p) := a /\!\!/^{A^0} b$$

and

$$(F', p') := (a \setminus^{A^0} b) \sqcup^{A^0} b.$$

Clearly, $F = F_1 \cup F_2 = (F_1 \setminus F_2) \cup F_2 = F'$. Moreover, $p = p'$ follows from simple case analysis. □

Moreover, some useful correspondences between $\setminus$ and $\sqcup$ and their set-theoretic counterparts exist:

11.5 Axiom
In A^0, the following equations hold for all cues a, b, and c:

$$(a \sqcup^{A^0} b) \setminus^{A^0} c = (a \setminus^{A^0} c) \sqcup^{A^0} (b \setminus^{A^0} c) \tag{11.7}$$

$$a \setminus^{A^0} (b \sqcup^{A^0} c) = (a \setminus^{A^0} b) \setminus^{A^0} c \tag{11.8}$$

$$(a \setminus^{A^0} b) \setminus^{A^0} c = (a \setminus^{A^0} c) \setminus^{A^0} b \tag{11.9}$$

$$(a \setminus^{A^0} b) \setminus^{A^0} c = (a \setminus^{A^0} (b \setminus^{A^0} c)) \setminus^{A^0} c \tag{11.10}$$

Proof *Let $(F_1, p_1) := a$, $(F_2, p_2) := b$, and $(F_3, p_3) := c$.*

(11.7): Let $(F, p) = (a \sqcup^{A^0} b) \setminus^{A^0} c$ and $(F', p') = (a \setminus^{A^0} c) \sqcup^{A^0} (b \setminus^{A^0} c)$. Then, $F = (F_1 \cup F_2) \setminus F_3 = (F_1 \setminus F_3) \cup (F_2 \setminus F_3) = F'$. Furthermore $p = p'$ follows from simple case analysis.

(11.8): Let $(F, p) = a \setminus^{A^0} (b \sqcup^{A^0} c)$ and $(F', p') := (a \setminus^{A^0} b) \setminus^{A^0} c$.

Now,

$$F = F_1 \setminus (F_2 \cup F_3) = (F_1 \setminus F_2) \setminus F_3 = F'.$$

Moreover, $p = p_{1|F} = p_{1|F'} = p'$ holds trivially.

(11.9): follows from (11.8) and Axiom 11.1.

(11.10): follows directly from the corresponding property of $\setminus$. □

11.6 Theorem
Restriction is associative in A^0. For all cues a, b, and c:

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

Proof

$$\begin{aligned}
a /\!\!/^{A^0} (b /\!\!/^{A^0} c) &= (a \setminus^{A^0} (b /\!\!/^{A^0} c)) \sqcup^{A^0} (b /\!\!/^{A^0} c) \\
\textit{(Theorem 11.4)} &= (a \setminus^{A^0} ((b \setminus^{A^0} c) \sqcup^{A^0} c)) \sqcup^{A^0} ((b \setminus^{A^0} c) \sqcup^{A^0} c) \\
\textit{(11.8)} &= ((a \setminus^{A^0} (b \setminus^{A^0} c)) \setminus^{A^0} c) \sqcup^{A^0} ((b \setminus^{A^0} c) \sqcup^{A^0} c) \\
\textit{(11.10)} &= ((a \setminus^{A^0} b) \setminus^{A^0} c) \sqcup^{A^0} ((b \setminus^{A^0} c) \sqcup^{A^0} c) \\
(\sqcup^{A^0} \textit{ associative}) &= (((c1 \setminus^{A^0} b) \setminus^{A^0} c) \sqcup^{A^0} (b \setminus^{A^0} c)) \sqcup^{A^0} c \\
\textit{(11.7)} &= (((a \setminus^{A^0} b) \sqcup^{A^0} b) \setminus^{A^0} c) \sqcup^{A^0} c \\
\textit{(Theorem 11.4)} &= (a /\!\!/^{A^0} b) /\!\!/^{A^0} c
\end{aligned}$$

□

11.7 Theorem
Restriction left-distributes over HTP in A_0: For all cues a, b, and c:

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

Proof

$$\begin{aligned}
(a \sqcup^{A^0} b) /\!\!/^{A^0} c &= ((a \sqcup^{A^0} b) \setminus^{A^0} c) \sqcup^{A^0} c \\
\textit{(11.7)} &= ((a \setminus^{A^0} c) \sqcup^{A^0} (b \setminus^{A^0} c)) \sqcup^{A^0} c \\
\textit{(Axiom 11.2)} &= ((a \setminus^{A^0} c) \sqcup^{A^0} (b \setminus^{A^0} c)) \sqcup^{A^0} c \sqcup^{A^0} c \\
\textit{(Axiom 11.2)} &= ((a \setminus^{A^0} c) \sqcup^{A^0} c) \sqcup^{A^0} ((b \setminus^{A^0} c) \sqcup^{A^0} c) \\
\textit{(Theorem 11.4)} &= (a /\!\!/^{A^0} c) \sqcup^{A^0} (b /\!\!/^{A^0} c)
\end{aligned}$$

□

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:

11.8 Axiom
For any factor μ and cues a and b, the following equations hold:

$$\begin{aligned}
\text{scale}(\mu, \text{black}) &= \text{black} && (11.11)\\
\text{scale}(\mu, a \sqcup b) &= \text{scale}(\mu, a) \sqcup \text{scale}(\mu, \text{b}) && (11.12)\\
\text{scale}(\mu, a /\!\!/ b) &= \text{scale}(\mu, a) /\!\!/ \text{scale}(\mu, \text{b}) && (11.13)\\
\text{scale}(\mu, a \setminus b) &= \text{scale}(\mu, a) \setminus b && (11.14)
\end{aligned}$$

11.5 An Algebraic Specification for Cues

The axioms of Section 11.4 form the basis for a simple algebraic specification of cues as an abstract data type. Figure 11.2 shows the result. The specification allows reasoning about cues without referring to their semantics.

11.9 Theorem
A^0 is a model for CUE0.

11.6 Cue Flat Form

Using arbitrary terms to represent cues is an effective way for the computer language expert to express and deal with look design. Terms—or trees with arbitrary depth—are not quite so easy to deal with for the ordinary user. Therefore, flat structures with a small number of levels are much more desirable than general terms. Fortunately, there is a language of flat cue terms which is able to express all general cue terms—the language of *cue flat forms*. There is only a small catch: the base language requires a slight modification to support the flat subset. This change complicates the language slightly, but actually clarifies the correspondence to the intuitive semantics somewhat.

The change becomes necessary through the nature of cue difference: to represent arbitrary differences, the flat language will require an additional *complement* operator. For a cue c, the complement yields another cue which contains exactly those fixtures not contained in c. Thus, subtracting the complement of c, $\overline{c}$, works rather like applying a stencil to a cue. Unfortunately, the complement does not

spec CUE0 ≡
signature Σ_{CUE0}
axioms
$\text{scale}(\mu, \text{black}) = \text{black}$
$\text{scale}(\mu, a \sqcup b) = \text{scale}(\mu, a) \sqcup \text{scale}(\mu, \text{b})$
$\text{scale}(\mu, a /\!\!/ b) = \text{scale}(\mu, a) /\!\!/ \text{scale}(\mu, \text{b})$
$\text{scale}(\mu, a \setminus b) = \text{scale}(\mu, a) \setminus b$
$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
$(a /\!\!/ b) /\!\!/ c = a /\!\!/ (b /\!\!/ c)$
$a \sqcup b = b \sqcup a$
$c \sqcup c = c$
$c /\!\!/ c = c$
$c \sqcup \text{black} = c$
$c /\!\!/ \text{black} = c$
$\text{black} /\!\!/ c = c$
$c \setminus \text{black} = c$
$\text{black} \setminus c = \text{black}$
$c \setminus c = \text{black}$
$(a \sqcup b) \setminus c = (a \setminus c) \sqcup (b \setminus c)$
$a \setminus (b \sqcup c) = (a \setminus b) \setminus c$
$(a \setminus b) \setminus c = (a \setminus c) \setminus b$
$(a \setminus b) \setminus c = (a \setminus (b \setminus c)) \setminus c$
$a /\!\!/ b = (a \setminus b) \sqcup b$
$(a \sqcup b) /\!\!/ c = (a /\!\!/ c) \sqcup (a /\!\!/ c)$
endspec

Figure 11.2: An algebraic specification for cues.

have a clear semantics on cues: What are the intensities of the fixtures in a complement? It should be unimportant, as complements should only occur as the second argument of the difference operator, but the language Σ_{CUE0} does not reflect that fact.

Figure 11.3 shows a new signature Σ_{CUE1} for cue terms which includes another sort *fixtureset* for sets of fixtures or *fixture sets*. They arise out of the semantics of cue difference: in its second argument, set difference only looks at the fixtures contained at as cue, not their intensities. Hence, in Σ_{CUE1}, difference accepts only fixtures sets as its second argument. An abstraction operator $\downarrow \_$ converts a cue to its associated set of fixtures, and the complement operates on fixture sets only.

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 simply extracts the first component from a cue:

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

The complement has the obvious set-theoretical interpretation:

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

Double complement is the identity on fixture sets:

signature $\Sigma_{\mathrm{CUE1}} \equiv$
sort *factor*
sort *fixture*
sort *fixtureset*
sort *cue*
functions
fromfixture : *fixture* $\rightarrow$ *cue*
scale : *factor*, *cue* $\rightarrow$ *cue*
black :$\rightarrow$ *cue*
$\_ \sqcup \_$: *cue*, *cue* $\rightarrow$ *cue*
$\_ /\!/ \_$: *cue*, *cue* $\rightarrow$ *cue*
$\downarrow \_$: *cue* $\rightarrow$ *fixtureset*
$\overline{\_}$: *fixtureset* $\rightarrow$ *fixtureset*
$\_ \setminus \_$: *cue*, *fixtureset* $\rightarrow$ *cue*
endsignature

Figure 11.3: A signature supporting cue flat forms.

11.10 Axiom
For any fixture set F, the following holds:

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

Proof □

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, abstraction will sometimes be implicit from here on. With this notational agreement, all axioms of Section 11.4 hold as before, and the axioms of CUE0 also apply after the insertion of the necessary abstraction operators. In A^1, another axiom holds:

11.11 Axiom
For cues a, b, and c, the following equation holds:

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

Proof *Let $(F_1, p_1) := a$, $(F_2, p_2) := b$, and $(F_3, p_3) := c$. Moreover, let $(F, p) = a \setminus^{A^1} (b \setminus^{A^1} c)$ and $(F', p') = (a \setminus^{A^1} b) \sqcup A^1(a \setminus^{A^1} \overline{c})$.*

The following equation holds by straightforward application of set theory:

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

Hence,

$$f \in F \Leftrightarrow f \in F_1 \wedge (f \notin F_2 \vee f \in F_3).$$

Also,

$$f \in F' \Leftrightarrow f \in F_1 \wedge (f \notin F_2 \vee f \in F_3).$$

Therefore, $F = F'$. Clearly, both p and p' are both restrictions of p_1 to $F = F'$. □

spec CUE1 $\equiv$
signature Σ_{CUE1}
axioms
$\text{scale}(\mu, \text{black}) = \text{black}$
$\text{scale}(\mu, a \sqcup b) = \text{scale}(\mu, a) \sqcup \text{scale}(\mu, \text{b})$
$\text{scale}(\mu, a /\!\!/ b) = \text{scale}(\mu, a) /\!\!/ \text{scale}(\mu, \text{b})$
$\text{scale}(\mu, a \setminus b) = \text{scale}(\mu, a) \setminus b$
$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
$(a /\!\!/ b) /\!\!/ c = a /\!\!/ (b /\!\!/ c)$
$a \sqcup b = b \sqcup a$
$c \sqcup c = c$
$c /\!\!/ c = c$
$c \sqcup \text{black} = c$
$c /\!\!/ \text{black} = c$
$\text{black} /\!\!/ c = c$
$c \setminus \downarrow \text{black} = c$
$\text{black} \setminus F = \text{black}$
$c \setminus \downarrow c = \text{black}$
$(a \sqcup b) \setminus F = (a \setminus F) \sqcup (b \setminus F)$
$a \setminus \downarrow (b \sqcup c) = (a \setminus \downarrow b) \setminus \downarrow c$
$(a \setminus F_1) \setminus F_2 = (a \setminus F_2) \setminus F_1$
$(a \setminus F_1) \setminus F_2 = (a \setminus (F_1 \setminus F_2)) \setminus F_2$
$a /\!\!/ b = (a \setminus \downarrow b) \sqcup b$
$(a \sqcup b) /\!\!/ c = (a /\!\!/ c) \sqcup (a /\!\!/ c)$
$\overline{\overline{F}} = F$
$a \setminus (b \setminus c) = (a \setminus b) \sqcup (a \setminus \overline{c})$
endspec

Figure 11.4: An algebraic specification for cues which differentiates cues from fixture sets.

Actually, making statements about difference would become easier with adding the missing operator from set theory—intersection. However, intersection does not have clear intuitive meaning when applied to lighting. Section 11.8 contains more discussion of this issue.

Figure 11.4 shows a revised algebraic specification CUE1 for cues based on Σ_{CUE1} which differentiates cues from fixture sets, and has the additional axioms.

It is now possible to define the flat cue language:

11.12 Definition
Assume that Σ_{CUE1} *has constants* $c_1, \ldots, c_n :\rightarrow$ *cue. An atom is either one of the* c_i *or a term of the form* $\text{fromfixture}(f)$ *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:

11.13 Theorem
Again assume that Σ_{CUE1} has constants $c_1, \ldots, c_n :\rightarrow$ cue. For each cue term t over Σ_{CUE1}, there exists a flat cue term t' equivalent to t in A^1. Moreover, it is possible to choose t' such that it contains the same atoms as t.

Proof *By term rewriting. The first objective to pull all HTP operators to the top level. The equations in Axiom 11.5 and Theorem 11.7 show how to do this for most cases. The remaining ones involve restriction which reduces to the previous cases by Theorem 11.4.*

HTP and restriction are associative, so that the internal structure of subterms involving only one of these two does not matter. This is not the case with differences. However, here Axiom 11.11 comes to the rescue. □

11.7 Algebra, Flat Form and User-Interface Design

The algebraic treatment of cue terms and the development of cue flat form are crucial prerequisites to developing an effective user interface to cues. This section only mentions the relevant aspects of the design. Please refer to Chapter 6 for details on Lula's actual user interface.

- Cue flat form means Lula can utilize a familiar user-interface construction for cue editing: Lula represents the upper level of cue flat forms—the "big HTP"—by a simple choice or tab widget. The second level has specialized editing widgets customized to the operator at hand.
- HTP and restriction are associative as per Axiom 11.1. This means the editing widget for HTP and restriction can use a linear list box.
- A left-associative sequence of differences

  $$c_0 \setminus c_1 \setminus \ldots \setminus c_n$$

  is commutative in $c_1, \ldots, c_n$ per Axiom 11.5. This again means that a list box with a specially treated editor for c_0 suffices to edit difference subterms in cue flat forms.

11.8 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 [Gunter and Scott, 1990]. From a conceptual standpoint, lighting works in a similar way: the more light there is on stage, the more is visible to the audience.[2]

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:

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

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

[2] This discounts the fact that strong back light can blind the audience and actually disclose *less* information at higher intensities. Ah well.

11.14 Theorem
$\sqsubseteq$ is a partial order.

Proof *reflexive: by commutativity of $\sqcup$.*
transitive: Consider cues a, b, and c with

$$a \sqsubseteq b \qquad b \sqsubseteq c.$$

This is equivalent to:

$$a \sqcup b = b \qquad b \sqcup c = c$$

Hence:

$$\begin{aligned} a \sqcup c &= a \sqcup (b \sqcup c) \\ &= (a \sqcup b) \sqcup c \\ &= b \sqcup c \\ &= c \end{aligned}$$

and therefore $a \sqsubseteq c$.
anti-symmetric: Consider two cues a and b with:

$$a \sqsubseteq b \qquad b \sqsubseteq a.$$

This means

$$a \sqcup b = b \qquad a \sqcup b = a$$

and thus $a = b$. □

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

11.15 Theorem
Cues form a complete partial order in A^1: every ω-chain in A^1_{cue} has a least upper bound.[3]

Proof *This result follows from the finiteness of $A^1_{fixture}$ and the boundedness of $\mathbb{I}$.* □

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

11.16 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 positively *removes* information from a cue.

Taking our formalization a little further. In A^1, $\sqcup$ induces a dual operation $\sqcap$:

$$(F_1, p_1) \sqcap^{A^1} (F_2, p_2) := (F_1 \cap F_2, p) \text{ where } p(f) := \min(a(f), b(f))$$

With this definition, $(A^1_{cue}, \sqsubseteq)$ even forms a complete distributive lattice. Unfortunately, it is not yet clear if this operator, if actually available to the operator for cue construction, would have any practical use. Presently, Lula does not include it.

Note also that fixture sets are a natural abstraction of cues in a domain-theoretic sense—cues and fixture sets form a Galois connection [Gunter, 1992] via the $\downarrow$ projection and a corresponding embedding $\uparrow$ with the following semantics:

$$\uparrow^{A^1} (F) := (F, p) \text{ where } p(f) := 0 \text{ for all } f \in F$$

[3]This is one of two common definitions for the term "complete partial order." This definition [Winskel, 1993], sometimes called "ω-cpo," is slightly weaker than the alternative "directed cpo" definition used in other places [Gunter and Scott, 1990].

11.9 Multi-Parameter Fixtures

The CUE1 specification is surprisingly specific to fixtures which allow intensity control only. Non-intensity parameters require a different 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.[4] 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.

11.10 A Semantic View of Multi-Parameters Cues

The domain-theoretic interpretation of cues presented in Section 11.8 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 brighter than 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 literally 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.

11.11 Modelling Parameter Transformations

Besides the issue of conflict, the introduction of multi-parameter fixtures raises a pragmatic problem: how does the concept of intensity scaling extend to the other parameters?

The previous chapter in Section 10.6 introduced the concept of *transformation* to generalize over changes in the setting of a certain parameter: a transformation maps a parameter setting to another parameter setting. Each transformation is specific to a certain parameter, and applies uniformly to all the fixtures in a cue.

Some of these transformations actually make use of an old setting to produce a new one—such as intensity scaling or applying an offset in stage coordinates—whereas others are simple value assignments. The former are called *relative* transformations, the latter *absolute*.

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

[4] Of course, a motor controlling one of the other parameter might be *audible*, but this problem is easy to fix in practice: put in a guitar solo or have an actor scream.

signature $\Sigma_{\text{TRAFO2}} \equiv$
sort *factor*
sort *itrafo*
sort *angle*
sort *pttrafo*
sort *trafo*
exception notrafo : *trafo*
functions
scale : *factor* $\rightarrow$ *itrafo*
$\epsilon_{itrafo} :\rightarrow$ *itrafo*
$\_ \circ_{itrafo} \_$: *itrafo*, *itrafo* $\rightarrow$ *itrafo*
$\_ \parallel \_$: *itrafo*, *itrafo* $\rightarrow$ *itrafo*
pan/tilt : *angle*, *angle* $\rightarrow$ *pttrafo*
$\epsilon_{pttrafo} :\rightarrow$ *pttrafo*
$\_ \circ_{pttrafo} \_$: *pttrafo*, *pttrafo* $\rightarrow$ *pttrafo*
$\_\#\_$: *itrafo*, *pttrafo* $\rightarrow$ *trafo*
$\_ \diamond \_$: *trafo*, *trafo* $\rightarrow$ *trafo*
$\_ \star \_$: *trafo*, *trafo* $\rightarrow$ *trafo*
endsignature

Figure 11.5: A signature for parameter transformations.

Composition arises when a cue already transformed 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. Figure 11.5 shows a signature for parameter transformations. It only supports parameters for intensity and pan/tilt. However, adding further parameters is analogous to pan/tilt. The signature differentiates between sorts for specific transformations for intensity and pan/tilt—*itrafo* and *pttrafo* and general transformations *trafo*.

Since juxtaposition is conceptually a partial function, ordinary algebraic specifications are not expressive enough. Some sort of exception handling is required. The Σ_{TRAFO2} signature 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. (It is coincidence that intensity transformations are relative in this specification, and pan/tilt absolute—other constructors are possible.) Moreover, ϵ_{itrafo} and $\epsilon_{pttrafo}$ are special constants meaning "no intensity (or pan/tilt, respectively) transformation specified."

The two $\circ$ compose intensity and pan/tilt transformations respectively. Moreover, $\parallel$ juxtaposes two intensity parameters.

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.

Figure 11.6 shows an algebraic specification for transformations matching the Σ_{TRAFO2} signature. In the specification, the propagation of the notrafo exception

spec TRAFO2 $\equiv$
signature Σ_{TRAFO2}
axioms
$\epsilon_{itrafo} \circ_{itrafo} i = i$
$i \circ_{itrafo} \epsilon_{itrafo} = i$
$\epsilon_{pttrafo} \circ_{pttrafo} p = p$
$p \circ_{pttrafo} \epsilon_{pttrafo} = p$
$(i_1 \# p_1) \diamond (i_2 \# p_2) = (i_1 \circ_{itrafo} i_2) \# (p_1 \circ_{pttrafo} p_2)$
$\epsilon_{itrafo} \parallel i = i$
$i_1 \parallel i_2 = i_2 \parallel i_1$
$(i_1 \# \epsilon_{pttrafo}) \star (i_2 \# p) = (i_1 \parallel i_2) \# p$
$t_1 \star t_2 = t_2 \star t_1$
$p_1 \neq \epsilon_{pttrafo}, p_2 \neq \epsilon_{pttrafo} \Longrightarrow (i_1 \# p_1) \star (i_1 \# p_2) = \text{notrafo}$
endspec

Figure 11.6: An algebraic specification for parameter transformations.

value is implicit [Klaeren, 1983, Gogolla *et al.*, 1984]. The error is not repairable, hence all variables are safe by assumption.

Finding an algebra A^2 for the TRAFO2 specification is straightforward: transformations are functions with special values for the ϵ constructors added.

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

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

The scale function has a new meaning in TRAFO2: it turns a factor into a function which scales an intensity value:

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

A pan/tilt setting consists of two angles. For example:

$$A^2_{angle} := \mathbb{R}^{0,+}_{\leq 2\pi}$$

The definition of $A^2_{pttrafo}$ is analogous to that of A^2_{itrafo}:

$$\begin{aligned} A^2_{pttrafo} &:= (A^2_{angle} \times A^2_{angle}) + \{\bot_{pttrafo}\} \\ \epsilon_{pttrafo} &:= \bot_{pttrafo} \end{aligned}$$

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:

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

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:

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

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, $trafo^{A^2}$ contains an exception element called $\lightning_{trafo}$:

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

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

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

Juxtaposition also works as in the specification:

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

11.12 Modelling Multi-Parameter Cues

The new signature for cues with pan/tilt fixtures is an extension of Σ_{TRAFO2}. Figure 11.7 shows it. The parts not related to the application of an intensity scale are largely unchanged from Σ_{CUE1}.

signature $\Sigma_{\text{CUE2}} \equiv$
extend $\Sigma_{\text{TRAFO2}} \cup \Sigma_{\text{BOOL}}$ **by**
sort *fixture*
sort *cue*
sort *setting*
functions
$\_@\_ : \mathit{fixture}, \mathit{trafo} \to \mathit{setting}$
$\_ \hookrightarrow \_ : \mathit{cue}, \mathit{setting} \to \mathit{bool}$
fromfixture : $\mathit{fixture} \to \mathit{cue}$
apply : $\mathit{trafo}, \mathit{cue} \to \mathit{cue}$
black : $\to \mathit{cue}$
$\_ \sqcup \_ : \mathit{cue}, \mathit{cue} \to \mathit{cue}$
$\_ /\!/ \_ : \mathit{cue}, \mathit{cue} \to \mathit{cue}$
$\downarrow \_ : \mathit{cue} \to \mathit{fixtureset}$
$\bar{\_} : \mathit{fixtureset} \to \mathit{fixtureset}$
$\_ \setminus \_ : \mathit{cue}, \mathit{fixtureset} \to \mathit{cue}$
endsignature

Figure 11.7: A signature for cues with pan/tilt fixtures.

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, Σ_{CUE2} has a new sort *setting*.

A *setting* is an association of a fixture with a transformation. In Σ_{CUE2}, it has, for a fixture f and a transformation t, the form $f@t$, pronounced "f is at t." The $\hookrightarrow$ operator relates a cue with a setting: For a cue c and 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."

Figure 11.8 shows an algebraic specification for cues matching Σ_{CUE2}. The rules for apply look much like the rules for scale in CUE0 and CUE1. However, there is an additional rule explaining the composition of transformations in terms of composition of its components.

The rules in Σ_{CUE2} 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 $\text{apply}(t, \text{fromfixture}(f))$. The other rules propagate settings up 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.

The other rules in CUE2 are identical to those in CUE1.

Building an algebra A^2 for CUE2 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_{cue} is a set with:

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

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

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

spec CUE2 $\equiv$
extend TRAFO2 $\cup$ BOOL **by**
signature Σ_{CUE2}
axioms

$\text{apply}(t, \text{black}) = \text{black}$
$\text{apply}(t, a \sqcup b) = \text{apply}(t, a) \sqcup \text{apply(t, b)}$
$\text{apply}(t, a /\!\!/ b) = \text{apply}(t, a) /\!\!/ \text{apply(t, b)}$
$\text{apply}(t, a \setminus b) = \text{apply}(t, a) \setminus b$
$\text{apply}(t_1, \text{apply}(t_2, c)) = \text{apply}(t_1 \diamond t_2, c)$

$\text{apply}(t, \text{fromfixture}(f)) \hookrightarrow f@t$
$a \hookrightarrow f@t_1 \wedge b \hookrightarrow f@t_2 \Longrightarrow (a \sqcup b) \hookrightarrow f@(t_1 \star t_2)$
$a \hookrightarrow f@t_1 \wedge \neg(\exists t_2.b \hookrightarrow f@t_2) \Longrightarrow (a \sqcup b) \hookrightarrow f@t_1$
$b \hookrightarrow f@t \Longrightarrow (a /\!\!/ b) \hookrightarrow f@t$
$a \hookrightarrow f@t_1 \wedge \neg(\exists t_2.b \hookrightarrow f@t_2) \Longrightarrow (a /\!\!/ b) \hookrightarrow f@t_1$
$a \hookrightarrow f@t_1 \wedge \neg(\exists t_2.b \hookrightarrow f@t_2) \Longrightarrow (a \setminus b) \hookrightarrow f@t_1$

$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
$(a /\!\!/ b) /\!\!/ c = a /\!\!/ (b /\!\!/ c)$
$a \sqcup b = b \sqcup a$
$c \sqcup c = c$
$c /\!\!/ c = c$
$c \sqcup \text{black} = c$
$c /\!\!/ \text{black} = c$
$\text{black} /\!\!/ c = c$
$c \setminus \downarrow \text{black} = c$
$\text{black} \setminus F = \text{black}$
$c \setminus \downarrow c = \text{black}$
$(a \sqcup b) \setminus F = (a \setminus F) \sqcup (b \setminus F)$
$a \setminus \downarrow (b \sqcup c) = (a \setminus \downarrow b) \setminus \downarrow c$
$(a \setminus F_1) \setminus F_2 = (a \setminus F_2) \setminus F_1$
$(a \setminus F_1) \setminus F_2 = (a \setminus (F_1 \setminus F_2)) \setminus F_2$
$a /\!\!/ b = (a \setminus \downarrow b) \sqcup b$
$a /\!\!/ b \sqcup b = a /\!\!/ c \sqcup b /\!\!/ c$
$\overline{\overline{F}} = F$
$a \setminus (b \setminus c) = (a \setminus b) \sqcup (a \setminus \overline{c})$

endspec

Figure 11.8: An algebraic specification for cues with pan/tilt fixtures.

The black cue has the same meaning as before:

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

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

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

The setting constructor @ is simple tupling:

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

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

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

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:

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

Restriction and difference basically work as before:

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

Relating settings to cues is straightforward in A^2:

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

11.17 Theorem
A^2 is a model of CUE2.

Proof *The axioms concerning $\hookrightarrow$ are the only ones requiring attention. The result by structural induction over cue terms:*

The base case is

$$\text{apply}^{A^2}(t, \text{fromfixture}^{A^2}(f)) \hookrightarrow (f,t).$$

Now, $\text{fromfixture}^{A^2}(f)$ *maps f to $(\bot_{itrafo}, \bot_{pttrafo})$ which is neutral with respect to $\diamond^{A^2}$. Therefore,*

$$\begin{aligned} (F,p) &:= \text{apply}^{A^2}(t, \text{fromfixture}^{A^2}(f)) \\ &= (\{f\}, \{f \mapsto t \diamond^{A^2} (\bot_{itrafo}, \bot_{pttrafo})\}) \\ &= (\{f\}, \{f \mapsto t\}) \end{aligned}$$

and consequently $(F,p) \hookrightarrow^{A^2} (f,t)$.

For the first non-base axiom, consider cues (F_1,p_1) and (F_2,p_2) and transformations t_1 and t_2 in A^2 as well as a fixture f with

$$(F_1,p_1) \hookrightarrow^{A^2} (f,t_1) \text{ and } (F_2,p_2) \hookrightarrow^{A^2} (f,t_2)$$

corresponding to the prerequisite of the second axiom for $\hookrightarrow$. Hence, $f \in F_1$ and $f \in F_2$. Let $(F,p) = (F_1,p_1) \sqcup A^2 (F_2,p_2)$ be non-exceptional. Therefore, by the definition of $\sqcup A^2$:

$$p(f) = p_1(f) \star^{A^2} p_2(f).$$

For cues (F_1,p_1) and (F_2,p_2), transformations t_1 and t_2 as well as a fixture f, the condition

$$\neg(\exists t_2 . b \hookrightarrow f@t_2)$$

simply means $f \notin F_2$. Again, by the definition of $\sqcup A^2$:

$$p(f) = p_1(f)$$

proving the axiom.

The specification has two rules for restriction. Restriction always prefers the transformation specified by the exponent if there is one. The prerequisite, for cues (F_1, p_1) and (F_2, p_2) and a transformation t corresponds to $f \in F_2$. The definition of restriction for $(F, p) := (F_1, p_1) /\!\!/^{A^2} (F_2, p_2)$ reduces to $p(f) = p_2(f)$, and, hence

$$(F, p) \hookrightarrow^{A^2} (f, t)$$

The prerequisite of the second restriction axiom translates to $f \notin F_2$. The definition yields $p(f) = p_1(f)$, proving the axiom. The difference axiom works in exactly the same way. □

11.13 Indirect Parameters and Fixture Calibration

CUE2 does not address the issue of indirect parameters—alternative representations of the ordinary "direct" parameters, such as RGB representations of color or Cartesian coordinates.

Ideally, the cue algebra would allow the transformation of indirect parameter as if they were direct ones. This is straightforward with an indirect parameter which has a fixed relationship with its direct counterpart. For example, the color wheels of color-changing units generally have fixed colors. Thus, translating an RGB representation of a color into the CMY and back as is necessary for driving the changer is a trivial matter.

On the other hand, some indirect parameters require installation-specific information for translation. Examples include finite-size effect wheels with exchangeable gels, and, more importantly, Cartesian-coordinate representation for a beam target. (Note that the mapping from to Cartesian coordinates to pan/tilt is not injective—a light beam crosses many points in space along its path.)

Loading the control system with the data necessary for performing the translation is called *calibration*. The system must provide a mapping from fixtures to the calibration information.

As long as every indirect parameter corresponds to exactly one direct parameter, the basic model of CUE2 remains unaffected. All that is necessary is

- a richer language for constructing transformations from mappings on indirect parameters, and
- providing the transformations with the calibration data upon application to a fixture.

The rest of the specification remains intact.

11.14 Implementation Notes

The implementation of cues in Lula strongly reflects the algebraic design and specifically cue flat form. A closer look at some aspects of the implementation illustrate the influence of the algebraic design. This section takes a bottom-up approach: more primitive data structures are first.

Flat Form Representation and Subcues The representation Lula uses for cue terms is restricted to flat forms. This ensures that the graphical user interface for edititing cues always sees and manipulates a valid representation. The important conceptual issue is the nature of the atomic components of cue flat forms.

In Lula, every cue has a name, and every cue term can reference another cue by that name. Specifically, the atomic subterms of cue flat form in Lula are so-called *subcues* consisting of a a cue and a transformation—the meaning of a subcue results from applying the transformation to the cue. A subcue is always part of exactly one *supercue*. Here is the representation for subcues:

```
(define-record-type :subcue
  (make-subcue cue supercue trafos)
  subcue?
  (cue subcue-cue!)
  (supercue subcue-supercue)
  (trafos subcue-trafos))
```

Lula internally uses the plural "trafos" to correspond to the *trafo* sort in the algebraic specification. This corresponds closely to the graphical representation—Lula displays each atomic components of a cue as a cue with a list of parameter transformations to its right. See Chapters 7 for some screen shots.

Representation of Transformations Lula assumes represents every parameter setting by a single Scheme value. A parameter transformation is essentially just a mapping between two settings of the same parameter. However, some metainformation is also needed as well as access to the fixture for calibration purposes. Therefore, Lula uses MzScheme's object system [Flatt, 2000] to implement a data-directed representation. The meat of the interface is this:

```
(define trafo<%>
  (interface ()
    get-aspect
    transform))
```

Note that even though the name is `trafo<%>`, the interface is only for single-parameter transformations, abstracting over the sorts *itrafo* or and *pttrafo* and others. `Get-aspect` returns the parameter a transformation is responsible for. (Parameters are called *aspects* in the implementation for historical reasons.) `Transform` takes a fixture and a parameter setting as arguments and returns a transformed setting. Here is the implementation for intensity-scaling transformations as an example:

```
(define intensity-scale-trafo%
  (class* object% (trafo<%>) (scale)
    (sequence
      (super-init))

    (public
     (get-aspect (lambda () intensity-aspect))
     (transform
      (lambda (fixture n)
        (* scale n)))
```

Complete transformations—members of CUE2's *trafo* sort are represented as lists of parameter transformations. When such a list lacks a certain aspect, the corresponding parameter transformation is assumed to be $\perp$.

Presets Lula stores the settings for a cue in a data structure called a *preset*. A preset is a mapping from fixture parameters to parameter values. This differs from the representation CUE2 and A^2 suggest where a setting is an association of a parameters to a transformations rather than a parameter value. Several reasons are behind the implementation choice:

- Both $CUE2$ and A^2 never address the issue of the actual parameter values—they do not need to. However, Lula ultimately *has* to produce the parameter values to drive the fixtures.
- Transformations are functions constructed by successive composition and juxtaposition. The efficient composition and juxtaposition of transformations requires structural knowledge the transformations do not naturally expose—`trafo<%>`'s `transform` is just a procedure. Representing the required structural information would be additional programming work and would limit the opportunities for abstraction.
- The structure of a transformation is only relevant as it relates to the cue structure. By itself, it has no useful meaning to the operator: only the resultant values are important.

Representing Cues Here, finally, is the type definition for cues. Note that its reactive aspects were already discussed in Section 9.4:

```
(define-record-type :cue
  (real-make-cue semaphore
                 name
                 flat-form
                 opt-preset upper-cues
                 structure-exchange preset-exchange)
  cue?
  (semaphore cue-semaphore)
  (name cue-name)
  (flat-form real-cue-flat-form unsafe-set-cue-flat-form!)
  (opt-preset real-cue-opt-preset unsafe-set-cue-opt-preset!)
  (upper-cues real-cue-upper-cues unsafe-set-cue-upper-cues!)
  (structure-exchange cue-structure-exchange)
  (preset-exchange cue-preset-exchange))
```

The semaphore synchronizes modifications to the mutable components of `:cue`. `Name` is the name of the cue. `Flat-form` is its term representation in flat form. `Opt-preset` is either the current preset of the cue of `#f`. The `upper-cues` component is a list of the upper cues—cues that include this one as a subcue. The two exchanges receive notifications from the cue subsystem upon structural changes to the cue itself as well as changes of its preset. The latter can also be caused by changes in the subcues.

Part VI

Lula in Motion

There's
somethin' wild in Lula I don't know
where it comes from.
—Marietta in *Wild at Heart*

A lighting console is a *animated reactive system*: It must continually drive (or *animate*) hardware attached to the console while reacting to internal and external events. Moreover, animation in lighting systems is *live*; reactivity must be instantaneous.

In conventional theater lighting, the variety of animations available to the operator is limited: Usually, the lighting during a theater production is static most of the time; it changes only for scene transitions with fades. However, concert and disco lighting often involve movement as part of the show itself.

Because of the complexity of modern multi-parameter fixtures, the operator must be able to use pre-programmed animations as well as design new ones: the operator effectively becomes a programmer.

Here is the biggest single challenge to designing a lighting control system: whereas the lighting designer thinks in terms of *what* should happen on stage, programming is usually concerned with *how* this happens.

This is especially obvious with simple systems, where operators commonly transcribe between conceptual movement ("make that moving light point at the drummer") and DMX512 slot values ("17 at 244, 18 at 12"). Complex animations such as smooth geometric figures are not possible with this level of control. Unfortunately, even sophisticated consoles subscribe to the latter user-interface philosophy, requiring the operator to describe an animation as a sequence of steps, with limited control over transition parameters, and even less control over interactive reactivity.

Consequently, to support sophisticated lighting animations, Lula goes a different route. The software uses *functional reactive animation* as a substrate to offer the user control over animated lighting. The key difference to traditional system is its declarative nature: the operator can concentrate on what she wants the animation to be, rather than how it should be implemented.

Lula equips the user with a simple domain-specific language for reactive animations called *Lulal*. The language grows with the user: simple animations only require mastery over very few language constructs. Sophisticated animations—which in scope go far beyond traditional consoles—involve abstraction, reactivity, and repetition.

Lula utilizes *Functional Reactive Programming* (FRP) for the description and implementation of all animated lighting. FRP focuses on separating the modelling from the rendering aspects of animation. Originally designed for graphics animations, FRP has applications in all fields involving continuous behaviors over time. Notable examples beside lighting include the construction of graphical user interfaces and robotics.

Chapter 12 introduces Functional Reactive Programming as a general concept and then describes the FRP framework which is part of Lula. Building on theses concepts, Chapter 13 shows the application of FRP to the animation of lighting and describes the Lulal language.

Chapter 12

Functional Reactive Programming

Hey, don't jump back so slow ... I thought you was a bunny ... Bunny jump fast—you jump back slow ... Mean somethin', don't it?
—Bobby Peru in *Wild at Heart*

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 [Elliott, 1997], graphical user interfaces [Sage, 2000], and robotics [Peterson *et al.*, 1999b, Peterson and Hager, 1999, Peterson *et al.*, 1999a]. As it turns out, it is also eminently applicable to animated lighting.

The advantage of using Functional Reactive Programming over more traditional imperative reactive frameworks [Boussinot, 1991, Boussinot and Susini, 1998, Pucella, 1998, Scholz, 1998] is its clear separation between modelling and presentation. FRP allows an implementor to provide a set of primitive reactive values and combinators to construct new ones which capture the essence of *what* an animation should be rather than *how* the application should render it. This makes FRP a classic declarative approach to a traditionally imperative problem.

The published implementations of FRP are written in Haskell [Haskell98, 1998], starting with Fran [Elliott and Hudak, 1997], a system for graphics animations. Implementations which provide the full declarative power of the approach—among them time transformation and recursive reactive values—inherently exploit functional programming and lazy evaluation. Unfortunately, these implementations basically assume that the entire program execution is under the control of the reactive sampling engine, and thus do not integrate well with existing program frameworks. Space leaks and the lack of synchrony complicate the use of Fran in realistic applications. Newer, imperative experimental implementations exist [Elliott, 1998d], but presently do not provide the full power of the functional implementations.

This chapter introduces FRP as a general concept, echoing some of Elliott's work [Elliott and Hudak, 1997, Elliott, 1998c]. It also describes Lula's subsystem for reactive programming which has some notable differences to traditional implementation approaches: Lula is written in Scheme, and the lack of implicit laziness requires more care in the implementation of the FRP primitives. Moreover, as Lula also allows hooking up user-interface components based on more traditional models of reactivity such as callbacks, it must provide *synchrony* between user actions and model reactions. This motivates an implementation strategy for reactive events

different from Fran's. Because documentation of many of the implementation details of FRP systems is generally sparse, this chapter provides some more technical insight into Lula's reactivity subsystem along with actual code for the most crucial combinators.

12.1 Reactive Values

Reactive programming requires a notion of time. There are two basic time-dependent entities in a reactive system [Elliott and Hudak, 1997]:

Behaviors represent time-varying values. Lula represents all lighting parameters, such as intensity, position, and color as behaviors. Even though a digital system will usually use discrete sampling to actually drive the fixtures, behaviors are conceptually continuous.

Events represent sources of *event occurrences*. An event occurrence represents the fact that something happened, usually along with some information on what it actually was that happened. The stream of button presses associated with a button can be a primitive event; the reactive aspects of other GUI components have similar representations as events. Events do not necessarily represent intervention from the outside world: alarm clocks and time-varying conditions are also primitive events.

Behaviors and events are not restricted to the primitive notions described above. Often, reactive values arise as transformations and combinations of other reactive values. Abstraction is common.

12.2 Semantics of Reactive Values

Functional reactive programming builds on a formal semantic model. This section presents a somewhat idealized semantics which provides a suitable intuition for understanding the nature of the reactive values in the framework.

The semantics uses a domain *time* for values representing points in time. To all intents and purposes, *time* values are real numbers.[1]

Behaviors and events both form parameterized abstract data types:

A behaviors in $behavior_\alpha$ represents time-varying value in α. The semantics given by the function **at** determines the value at a certain amount in time. To do this, **at** takes *two* points in time as an argument: the second one is the time of interest for which **at** determines the value. The first time argument is the *start time*. The start time is relevant for reactive behaviors which change according to occurring events. Such a behavior takes only event occurrences into account happening after the start time.

$$\mathbf{at} : behavior_\alpha \to time \times time \to \alpha$$

Events map an interval in time to a sequence of event occurrences—pairs of values and timestamps.

$$\mathbf{occs} : event_\alpha \to time \times time \to (\alpha \times time)^*$$

For an event e, **occs**(e) returns a function accepting two points in time, t_1 and t_2, representing an interval $(t_1, t_2]$. The function returns a finite sequence of time-ascending occurrences in that interval. Note that **occs** does not catch occurrences happening precisely at t_1.

[1] A precise interpretation also includes *partial elements* [Elliott and Hudak, 1997]. This view does little to enhance the intuition, however.

12.2.1 Constructing Simple Behaviors

Simple behaviors result by the construction of primitive behaviors from ordinary values and functions over their domains, and the primitive time behavior.

Time The most primitive behavior is time itself. Its semantics is trivial:

$$\begin{aligned} \text{time} &: \mathit{behavior}_{\mathit{time}} \\ \mathbf{at}(\text{time}) &:= \lambda(t_s, t).t \end{aligned}$$

Lifting The simplest constructed behavior is probably a constant one. The process converting a value into a behavior is called *lifting*:

$$\begin{aligned} \text{lift} &: \alpha \to \mathit{behavior}_{\alpha} \\ \mathbf{at}(\text{lift}(v)) &:= \lambda(t_s, t).v \end{aligned}$$

Lifting generalizes to arbitrary functions between "ordinary values": Lifting converts a function from values to a value into a function from behaviors to a behavior. For $n \in \mathbb{N}$, the lifting operator for an n-ary function is lift_n:

$$\begin{aligned} \text{lift}_n &: ((\alpha_1 \times \cdots \times \alpha_n) \to \beta) \to (\mathit{behavior}_{\alpha_1} \times \cdots \times \mathit{behavior}_{\alpha_n}) \to \mathit{behavior}_{\beta} \\ \mathbf{at}(\text{lift}_n(f)(b_1, \ldots, b_n)) &:= \lambda(t_s, t).f(\mathbf{at}(b_1)(t_s, t), \ldots, \mathbf{at}(b_n)(t_s, t)) \end{aligned}$$

Note that $\text{lift}_0 = \text{lift}$.

Time Transformation Time transformation is one of the most attractive features of the framework: it replaces a behavior's notion of time with another, itself denoted by a behavior.

$$\begin{aligned} \text{time-transform} &: \mathit{behavior}_{\alpha} \times \mathit{behavior}_{\mathit{time}} \to \mathit{behavior}_{\alpha} \\ \mathbf{at}(\text{time-transform}(b, b_t)) &:= \lambda(t_s, t).\, \mathbf{at}(b)(t_s, \mathbf{at}(b_t)(t_s, t)) \end{aligned}$$

12.2.2 Constructing Events

Primitive events come from two sources: foreknowledge of occurrence time and value, and external sources controlled by the user.

Constant Events The simplest kind of event has only one occurrence. It is like an old-fashioned alarm clock:

$$\begin{aligned} \text{alarm} &: \mathit{time} \times \alpha \to \mathit{event}_{\alpha} \\ \mathbf{occs}(\text{alarm}(t, v)) &:= \lambda i.[(t, v) \mid t \in i] \end{aligned}$$

A similar construct is closer to modern digital watches:

$$\begin{aligned} \text{periodic-alarm} &: \mathit{time} \times \mathit{dtime} \times \alpha \to \mathit{event}_{\alpha} \\ \mathbf{occs}(\text{periodic-alarm}(t, \delta, v)) &:= \lambda i.[(t + n\delta, v) \mid n \in \mathbb{N}, t + n\delta \in i] \end{aligned}$$

External Events External sources of event occurrences are typically UI elements such as the keyboard, or GUI controls. The semantics assumes that these have primitive events associated with them, such as:

$$\begin{aligned} \text{keyboard} &: \mathit{event}_{\mathit{char}} \\ \text{left-mouse-button} &: \mathit{event}_{\mathit{unit}} \\ \text{mouse-position} &: \mathit{event}_{\mathbb{R} \times \mathbb{R}} \end{aligned}$$

Event Handling New events result from old ones primarily through *event handling*—transforming an event into another one whose occurrences happen at the same points in time, but with different resultant values. Ideally, the function performing the transformation will have access to both time and value of every occurrence:

$$\begin{aligned} \text{handle-event} &: \mathit{event}_\alpha \times (\mathit{time} \times \alpha \to \beta) \to \mathit{event}_\beta \\ \mathbf{occs}(\text{handle-event}(e, f)) &:= \lambda i.\,[(t_i, f(t_i, v_i)) \mid \mathbf{occs}(e)i = [(t_1, v_1), \ldots, (t_k, v_k)]] \end{aligned}$$

Handle-event induces a number of derived combinators which do only part of its work:

$$\begin{aligned} \text{map-event} &: \mathit{event}_\alpha \times (\alpha \to \beta) \to \mathit{event}_\beta \\ \text{map-event}(e, f) &:= \text{handle-event}(e, \lambda(t, v).fv) \\ \text{time-only-event} &: \mathit{event}_\alpha \times (\mathit{time} \to \beta) \to \mathit{event}_\beta \\ \text{time-only-event}(e, f) &:= \text{handle-event}(e, \lambda(t, v).ft) \\ \text{const-event} &: \mathit{event}_\alpha \times \beta \to \mathit{event}_\beta \\ \text{const-event}(e, v^*) &:= \text{handle-event}(e, \lambda(t, v).v^*) \end{aligned}$$

Filtering Events Sometimes, it is desirable to filter out the event occurrences of an event satisfying a given predicate:

$$\begin{aligned} \text{filter} &: \mathit{event}_\alpha \times (\alpha \times \mathit{time} \to \mathit{boolean}) \to \mathit{event}_\alpha \\ \mathbf{occs}(\text{filter}(e, p)) &:= \lambda i.[(t_i, v_i) \mid \mathbf{occs}(e)i = [(t_1, v_1), \ldots, (t_k, v_k)], p(t_i, v_i)] \end{aligned}$$

Predicate Events A *predicate event* watches a boolean behavior and produces an occurrence every time the behavior changes from *false* to *true*. Whereas the intuition is simple, the semantics is somewhat involved:

$$\text{predicate} : \mathit{behavior}_{\mathit{boolean}} \to \mathit{event}_{\mathit{unit}}$$

For the semantics of predicate(b) and an interval $i := (t_a, t_b)$, consider a partition $[t0, \ldots, t_n]$ of $[t_a, t_b]$ values $c_1, \ldots, c_n \in \mathit{boolean}$ compatible with b in the following way:

- For all $j \in \{1, \ldots, n-1\}$, $c_j = \neg c_{j+1}$.
- For all $j \in \{1, \ldots, n-1\}$, $t \in (t_{j-1}, t_j)$ implies $\mathbf{at}(b)t = c_i$.

If such a partition exists, then $\mathbf{occs}(\text{predicate}(b))i := o$ where o is the shortest time-ascending sequence with the following elements:

- If $c_1 = \mathbf{true}$ $\mathbf{at}(b)t_a = \mathbf{false}$, then $(t_a, ()) \in o$.
- For $j \in \{1, \ldots, n-1\}$, $(t_j, ()) \in o$ if $c_j = \mathbf{false}$.
- If $c_n = \mathbf{false}$ and $\mathbf{at}(b)t = \mathbf{true}$, then $(t, ()) \in o$.

Choice The merge operator combines the occurrences of two compatible events into one:

$$\begin{aligned} \text{merge} &: \mathit{event}_\alpha \times \mathit{event}_\alpha \to \mathit{event}_\alpha \\ \mathbf{occs}(\text{merge}(e_1, e_2)) &:= \lambda i.[(t_i, v_i) \mid (t_i, v_i) \in \mathbf{occs}(e_1)i \text{ or } (t_i, v_i) \in \mathbf{occs}(e_2)] \end{aligned}$$

This definition is actually ambiguous: If e_1 and e_2 both have occurrences at the exact same time, the definition does not specify which one merge specifies in the output sequence. Ideally, the occurrences in e_1 would take precedence. The difference is unimportant for the most purposes.

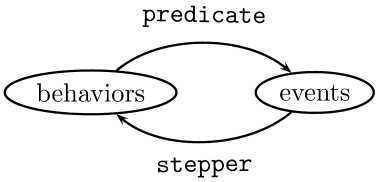

Figure 12.1: Converting from behaviors to events and back.

12.2.3 Combining Behaviors and Events

A number of combinators take both events and behaviors as parameters and combine them in various ways.

Reactivity The key combinator for constructing behaviors from events is until: it behaves like one behavior until an event occurs, and then switches to behaving like a different behavior produced by the event:

$$\begin{aligned} \text{until} &: \mathit{behavior}_\alpha \times \mathit{event}_{\mathit{behavior}_\alpha} \to \mathit{behavior}_\alpha \\ \mathbf{at}(\text{until}(b,e))(t_s,t) &:= \begin{cases} \mathbf{at}(b)(t_s,t) & \text{if } \mathbf{occs}(e)(t_s,t) = [] \\ & \text{or } \mathbf{occs}(e)(t_s,t) = [(t_1,b_1),\ldots] \text{ and } t \le t_1 \\ \mathbf{at}(b_1)(t_1,t) & \text{otherwise for } \mathbf{occs}(e)(t_s,t) = [(t1,b_1),\ldots] \end{cases} \end{aligned}$$

Converting Events to Behaviors An event has a natural analogue as a piecewise-constant behavior, where the value at a point in time is determined by the value of the last event occurrence:

$$\begin{aligned} \text{stepper} &: \alpha \times \mathit{event}_\alpha \to \mathit{behavior}_\alpha \\ \mathbf{at}(\text{stepper}(v,e)) &:= \lambda(t_s,t).\begin{cases} v & \text{if } \mathbf{occs}(e)(t_s,t) = [] \\ v_j & \text{if } \mathbf{occs}(e)(t_s,t) = [(t_1,v_1),\ldots,(t_n,v_n)], t_j < t \text{ and } t_j + 1 \ge t \\ v_n & \text{if } \mathbf{occs}(e)(t_s,t) = [(t_1,v_1),\ldots,(t_n,v_n)] \text{ and } t_n < t \end{cases} \end{aligned}$$

Figure 12.1 illustrates the correspondence between behaviors and events established by `predicate` and `stepper`.

Sampling Behaviors Sometimes it is useful to obtain the value of a behavior at the time an event occurs. The snapshot does this:

$$\begin{aligned} \text{snapshot} &: \mathit{event}_\alpha \times \mathit{behavior}_\beta \to \mathit{event}_{\alpha\times\beta} \\ \mathbf{occs}(\text{snapshot}(e,b)) &:= \lambda(t_s,t).[(t_1,(a_1,b_1)),\ldots,(t_n,(a_n,b_n))] \\ &\quad \text{where } \mathbf{occs}(e)(t_s,t) = [(t_1,a_1),\ldots,(t_n,a_n)] \text{ and } \mathbf{at}(b)(t_s,t_j) = b_j \end{aligned}$$

12.2.4 Integral and Differentiation

Integration and differentiation are also possible:

$$\begin{aligned} \text{integrate} &: \mathit{behavior}_{\mathit{real}} \to \mathit{behavior}_{\mathit{real}} \\ \mathbf{at}(\text{integrate}(b)) &:= \lambda(t_s,t).\int_{t_s}^{t} (\mathbf{at}(b)(t_s,\tau))d\tau \\ \text{derivative} &: \mathit{behavior}_{\mathit{real}} \to \mathit{behavior}_{\mathit{real}} \\ \mathbf{at}(\text{derivative}(b)) &:= \lambda(t_s,t).\left(\frac{\mathbf{at}(b)(t_s,\tau)}{d\tau}\right) \end{aligned}$$

Implementation use numerical approximation. Especially integration is extremely useful in, for instance, the simulation of physical behavior. In Lula, an integral behavior defines progress in a light fade, dependent on the position of the "Speed" fader.

12.3 Implementation Issues

The semantics given in Section 12.2 suggests modelling the representation of behaviors and events after **at** and **occs**: the semantics maps both to functions—in a functional language, these have first-class representations. There are two pragmatic problems, however:

- The semantics allow looking arbitrarily far back into the past: both **at** and **occs** produce functions which accept arbitrary times as arguments. Even though simple behaviors have representations as simple functions, truly reactive values would have to record all event occurrences forever to support the full semantics.

  This is not just a space problem: the functional semantics do not easily allow for any bookkeeping between samplings: reactive values involving, say, cascaded applications of until, must re-sample all events of the past at every sampling interval. Some caching could remedy the problem, at the cost of an even bigger space leak [Elliott, 1998c].

- The semantics allow looking arbitrarily far into the future, again because there are no constraints on the sampling times. Reactive programming must be able to function in an interactive environment where the system does not know all event occurrences in advance.

Primarily because of these two problems, implementations of FRP have used a variety of representations different from the naive one suggested by the semantics [Elliott, 1998b, Elliott, 1998c, Elliott, 1998d].

All representations must tackle the problem of *aging*: the running application must—implicitly or explicitly—allow the FRP substrate to "forget" events that happen in the past. There are fundamentally two ways of doing this:

- Use an applicative representation and automatic storage management to reclaim space for old events no longer needed [Elliott, 1998b, Elliott, 1998c]. When necessary, require the programmer to provide aging annotations.

- Use an imperative representation [Elliott, 1998d] which observes only "current time."

The two approaches to the representation of reactive values differ along several axes. The most fundamental one seems to be that declarative representations are demand-driven and thus present a pull-based notion of reactivity, whereas imperative implementations are push-based: event occurrences drive progress in the system (see Section 9.3.1).

The decision for one or the other—besides having various consequences for the programmer—is visible to user: push-based implementations usually exhibit *synchronicity* between an event occurrence and the reaction to it, whereas pull-based systems usually have sampling loops which notice event occurrences only at sampling intervals. Depending on the duration of the sampling interval, the delay may be quite visible to the user.

Hence, Lula uses a hybrid representation which strives to retain the good declarative properties of the applicative representation, while supporting synchronicity

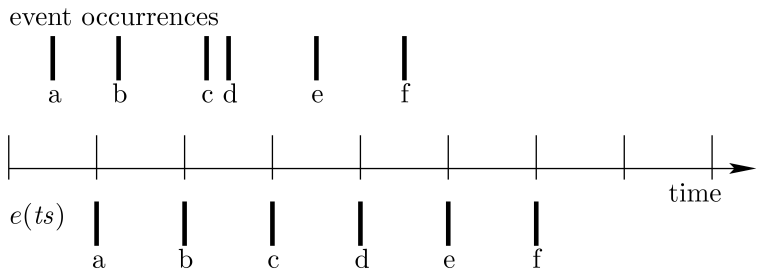

Figure 12.2: Actual event occurrences and sampling.

where desired. The central problem that any implementation of FRP must address is that of sampling *event non-occurrences*. Lula's FRP implementation is closely related to a simple stream-based implementation of FRP.

12.4 Stream-Based Reactive Values

Consider a stream-based representation of behaviors and events [Elliott, 1998b, Elliott, 1998c]. The stream-based representation has the advantages of being suitable for realistic implementation, while having a closely, formally explored relationship to the semantics [Wan and Hudak, 2000].

For now, a *stream* is a—potentially infinite—sequence of values. A stream of values in α is denoted by $stream_\alpha$. Here are the representations on behaviors and events:

$$\begin{aligned} behavior_\alpha &:= stream_{time} \rightarrow stream_\alpha \\ event_\alpha &:= stream_{time} \rightarrow stream_{opt_\alpha} \end{aligned}$$

(opt_α stands for a value in α or for some special value ϵ denoting "nothing.") Both representations map monotonically increasing time streams (a constraint not expressed in the domains) to other streams, the idea being that the elements in the input stream correspond to elements in the output stream.

Hence, the function representing a behavior takes a stream of sampling times as its argument and returns a stream of the values of the behavior at those times. For events, the situation is slightly more complicated: for a sampling time t in the input stream, the corresponding optional value in the output stream says whether there was an event occurrence *before* t. When there are more than two event occurrences between two sample times, the occurrences reported in the output stream distribute over the following elements. Figure 12.2 illustrates the situation. Thus, the underlying assumption is that, over time, sampling will happen more often than event occurrence.

Note that it is crucial that the semantics be able to express non-occurrence explicitly: the output stream will contain *epsilon* if the event did not occur between the previous sample time and the current one. The sampling of reactive values, such as behaviors created by until, will need to know for any given time t, if the event has occurred before t.

Since the stream-based representation offers only an approximation of the true semantics, the original definitions of **at** and **occs** are no longer appropriate. More reasonable semantics for these representations are the two functions $\tilde{\mathbf{at}}$ and $\tilde{occs}$

which accept finite time streams rather than intervals:

$$\begin{aligned} \tilde{\mathbf{at}} &: behavior_\alpha \to stream_{time} \to \alpha \\ \tilde{\mathbf{at}}(b) &:= \lambda ts.\, \mathrm{last}(b(ts)) \\ \mathbf{o\tilde{c}cs} &: event_\alpha \to stream_{time} \to stream_{time \times \alpha} \\ \mathbf{o\tilde{c}cs}(e) &:= \lambda ts.[(t,v) \mid (t,v) \in e(ts), v \neq \epsilon] \end{aligned}$$

The last function merely extracts the last element of a finite stream.

Under certain constraints, $\tilde{\mathbf{at}}$ and $\mathbf{o\tilde{c}cs}$ approximate the **at** and **occs** functions, respectively [Wan and Hudak, 2000]. Of course, certain combinators like predicate which depend on the ability to catch a change in a continuous behavior can miss instantaneous conditions. (Other implementations of FRP allow *interval analysis* which can detect instantaneous peaks in behaviors [Elliott and Hudak, 1997]; they are less suited for practical implementation, however.)

12.5 Implementation of Reactive Values

Lula represents every source of output to the interface as a behavior over presets. This requires smooth integration of FRP with the rest of Lula, a stateful system which needs to run reliably for extended periods of time. This requirement imposes special constraints on the implementation:

1. Even though the implementation must sometimes remember event occurrences in the past, it must not run out of memory. This is the most serious issue.

2. The system must be able to repeatedly start and stop sampling of behaviors as it runs. This is different from closed-world systems like Frob and Fran where one application essentially samples one single behavior forever.[2]

3. The system must function in a multi-threaded environment; it must allow integration with traditional callback-based GUI systems.

12.5.1 Streams, Laziness and Recursion

Lula's implementation of FRP is essentially stream-based, just like the representation described in Section 12.4. Since Lula is written in Scheme, a strict language, the most immediate question is how to represent streams.

Ordinary (strict) lists will not do since the streams which occur in FRP are not completely available when computation starts and potentially become infinite. Representing lazy lists in Scheme is easy enough through `delay` and `force` which implement delayed evaluation. However, for non-strict lists, there are two different representations with different degrees of strictness [Wadler *et al.*, 1998]:

- The *odd style* of representing lazy lists uses `delay` for the `cdr` of the list. The lazy list corresponding to the stream $[1, 2]$ is constructed in the following way:

  ```
  (cons 1 (delay (cons 2 (delay '()))))
  ```

  This style of constructing lazy lists seems fairly common in the Scheme community [Abelson *et al.*, 1996]. This style is called "odd" because a finite lazy list contains an odd number of constructors, counting `cons`, `delay`, and `'()` [Wadler *et al.*, 1998]. Note that the representation is strict in the head of the list.

[2] This can actually turn into an advantage: The current implementation of Fran still leaks memory, partly because it holds on to the history of the user interactions for the entire runtime of the program.

- The *even style* moves the `delay` operator to the front. Here is the representation for $[1, 2]$:

  ```
  (delay (cons 1 (delay (cons 2 (delay '())))))
  ```

  With this style, the number of constructors is always even. Whereas the even style leads to somewhat more convoluted programs, it yields the behavior which programmers familiar with non-strict languages usually expect.

(The problem is not prominent in the published literature on FRP, as it exclusively uses Haskell [Haskell98, 1998], a non-strict language, where *everything* is lazy.)

Laziness is an important issue with recursively defined reactive values such as integral (see Section 12.5.7). Simultaneous behavior and event values might have interdependencies, requiring delayed evaluation for both. Consequently, Lula's reactivity kernel uses the even style for lazy lists:

```
(define-syntax stream-cons
  (syntax-rules ()
    ((stream-cons head tail)
     (delay (cons head tail)))))

(define (stream-car stream)
  (car (force stream)))

(define (stream-cdr stream)
  (cdr (force stream)))

(define (stream-null? stream)
  (null? (force stream)))
```

Actually, for events, the reactivity library further protects the lists with semaphores to allow concurrent access in a multithreaded setting, but the strictness behavior remains the same.

The implementations for behaviors and events used by Lula's reactivity subsystem allow the definition of two procedures `at` and `occurrences` which return the applicative representation presented in Section 12.4:

`(at` *behavior time-list*`)` procedure[3]

`At` accepts a behavior and a lazy list of sample times, and returns a lazy list of values of the behaviors.

`(occurrences` *event time-list*`)` procedure

`Occurrences` accepts an event and a lazy list of sample times, and returns a lazy list of possible occurrences which are `#f` for non-occurrence or a record consisting of a timestamp and the promise of a value:

```
(define-record-type :occurrence
  (make-occurrence time promise)
  occurrence?
  (time occurrence-time)
  (promise occurrence-promise))
```

[3]The description of Scheme procedures or macros uses the same format as the R^{5}RS [Kelsey *et al.*, 1998]. "Procedure" in this case means that `at` is a procedure, not a macro.

12.5.2 Implementing Behaviors

The straightforward stream-based implementation of behaviors as functions from time streams to value streams is simple enough, but has at least one fundamental problem which makes it unsuitable for use in long-running systems:

Consider a reactive behavior $b := \text{until}(b', e)$ for an arbitrary behavior b and an event e. As long as the program accesses this behavior, it also needs access to e. Now, even though b depends only occurrence of e, e may in fact have many other occurrences, all of which must stay accessible and therefore consume memory. Specifically, if e is tied to some UI component—say, a mouse button—it will accumulate more occurrences as run time progresses, all of which have to be stored. This is a space leak, and the representation does not allow for a fix.

Hence, Lula's reactivity subsystem uses a richer inductive representation for behaviors which, for examples, allows discarding e in the example above after its first occurrence. It is close, but not identical to the representation used in Fran [Elliott, 1998b, Elliott, 1998c, Elliott, 1999, Elliott, 1998a].

The representation is derived from the set of behavior constructors. It consists of a number of record types whose names start with `behavior*`. Note that this is only the representation of the behavior structure. The actual representation for behaviors, `behavior`, is a promise of a `behavior*:...` value to allow for recursion.

There is are two truly primitive behaviors, constant behaviors and time:

```
(define-record-type :behavior*:constant
  (make-behavior*:const value)
  behavior*:const?
  (value behavior*:const-value))

(define-record-type :behavior*:time
  (make-behavior*:time)
  behavior*:time?)
```

These two have straightforward semantics expressed by a procedure `at*` which, for a behavior structure `behavior*` maps a time stream to a value stream:

```
(define (at* behavior* time-stream)
  (cond
   ((behavior*:time? behavior*) time-stream)
   ((behavior*:constant? behavior*)
    (repeat-stream (behavior*:constant-value behavior*)))
```

`Repeat-stream` just repeats its argument:

```
(define (repeat-stream x)
  (stream-cons x (repeat-stream x)))
```

Another type of behavior is the arises from the lifted version of function application: a behavior of functions from values to values is applied to a behavior of values:

```
(define-record-type :behavior*:app
  (make-behavior*:app proc-behavior arg-behavior)
  behavior*:app?
  (proc-behavior behavior*:app-proc-behavior)
  (arg-behavior behavior*:app-arg-behavior))
```

The semantics of `app` behaviors results from taking the semantics of both the `proc-behavior` and the `arg-behavior` component and applying them elementwise to each other. The `at` procedure used in the semantics appears further below. Note that this is a continuation of the `cond` expression that is the body of `at*`:

```
   ((behavior*:app? behavior*)
    (streams-apply (at (behavior*:app-proc-behavior behavior*) time-stream)
                   (at (behavior*:app-arg-behavior behavior*) time-stream)))
```

`Streams-apply` procedure has the obvious definition:

```
(define (streams-apply proc-stream arg-stream)
  (stream-cons ((stream-car proc-stream) (stream-car arg-stream))
               (streams-apply (stream-cdr proc-stream)
                              (stream-cdr arg-stream))))
```

Time-transformed behaviors also have their own place in the representation:

```
(define-record-type :behavior*:time-transformed
  (make-behavior*:time-transformed behavior time-behavior)
  behavior*:time-transformed?
  (behavior behavior*:time-transformed-behavior)
  (time-behavior behavior*:time-transformed-time-behavior))
```

Its semantics again uses `at`, in a straightforward way:

```
   ((behavior*:time-transformed? behavior*)
    (at (behavior*:time-transformed-behavior behavior*)
        (at (behavior*:time-transformed-time-behavior behavior*)
            time-stream)))
```

The most significant case—as far as space behavior is concerned—is the `behavior*:until` type:

```
(define-record-type :behavior*:until
  (make-behavior*:until behavior event)
  behavior*:until?
  (behavior behavior*:until-behavior)
  (event behavior*:until-event))
```

Implementing the semantics of `behavior*:until` involves actually sampling it. To this end, `at*` uses `at` and `occurrences` to generate parallel lazy lists of behavior values and possible occurrences. Sampling loops over both lazy lists until it finds an event occurrence. As soon as that happens, `at` switches to the behavior stashed in the event occurrence:

```
   ((behavior*:until? behavior*)
    (delay
      (let loop ((time-stream time-stream)
                 (values (at (behavior*:until-behavior behavior*) time-stream))
                 (maybe-occurrences (occurrences (behavior*:until-event behavior*)
                                                 time-stream)))
        (let ((maybe-occurrence (stream-car maybe-occurrences)))
          (if maybe-occurrence
              (force (at (force maybe-occurrence) time-stream))
              (cons (stream-car values)
                    (delay
                      (loop (stream-cdr time-stream)
                            (stream-cdr values)
                            (stream-cdr maybe-occurrences)))))))))
```

The code above is a slightly simplified version of the code actually in Lula—it is a little too strict in forcing the event value. This matters in recursive reactive values;

there is a more detailed explanation of the mechanisms involved in Section 12.5.7 further below.

To support recursion, the behavior representation wraps the structural information represented by the `behavior*:...` types in a promise:

```
(define-record-type :behavior
  (make-behavior *-promise)
  behavior?
  (*-promise behavior-*-promise))
```

Thus, the `at` procedure that computes the semantics of behaviors uses a `delay-force` pair to unwrap/wrap the promise of the behavior structure:

```
(define (at behavior time-stream)
  (delay (force (at* (force (behavior-*-promise behavior)) time-stream))))
```

The representation for behaviors is similar to that used in Fran [Elliott, 1998b, Elliott, 1998c, Elliott, 1999, Elliott, 1998a]. Fran's representation is somewhat more involved, mainly because Haskell, the language Fran is written in, presently cannot express the types for `behavior*:time` and `behavior*:app`. Moreover, Fran uses memoization to avoid redundant sampling [Elliott, 1998b, Elliott, 1998c], but it is not clear whether the added implementation complexity yields any significant performance improvement. If fact, caching sometimes actually slows down sampling.

12.5.3 Event Non-Occurrences and Synchrony

Whereas the implementation of behaviors arises in a straightforward way from the stream-based representation, the implementation of events is more involved. Again, just as with behaviors, representing an event as a stream-transforming functions does not allow for a reasonable form of garbage collection.

Fran instead uses an explicit first-oder representation—an event is a stream of possible occurrences. However, Fran's representation for events raises two immediate issues: *blocking* and *synchrony*. This section discusses Fran's approach to the problem and the blocking issue; the following section describes Lula's implementation which differs from Fran's.

Here is Fran's representation for events:

$$event_\alpha := stream_{time \times opt_\alpha}$$

Again, the stream of event occurrences representing an event must be monotonically increasing with respect to the timestamps.

The stream may also contain elements containing only a timestamp. An element (t, ϵ) represents a *non-occurrence*: it means that the event did not occur between the timestamp of the previous element in the stream and t. This may surprising at first—an event

$$[(0, a), (1, \epsilon), (2, b), (3, \epsilon), (4, \epsilon), (5, c)]$$

is obviously equivalent to one without the non-occurrences:

$$[(0, a), (2, b), (5, c)]$$

However, in an interactive application, most event occurrences are typically not known when the system starts—they arise from interaction with the user over time. If the stream representing an interactive event could not contain occurrences, accessing, say, the first element of the stream might block until the interaction actually occurs. This is unacceptable: the animation must be able to continue; accessing the stream representing an event must never block. Therefore, when the sampling loop

accesses an event that is still waiting to happen, the event stream will generate a non-occurrence.

The necessity of representing non-occurrences exposes a deeper issue with the practical implementation of FRP: sampling a reactive value at a given time can only take into account what event occurrences are known *at the time of sampling*. In fact, it is entirely possible, that, through communication latencies, an event occurrence becomes known at a wallclock time significantly later than the time of the occurrence itself. This breaks the monotonicity requirement for the occurrence stream. Fortunately, this is not a problem in practice as long as the actual occurrences in the stream are in the right order. The system will still react to the occurrence, albeit with a delay. (Of course, the delay may actually cause non-continuous changes in the animation, but there is not really a way to rectify the problem when the communication of event occurrences is not instantaneous.)

The important insight is that the system will usually generate non-occurrences for the current wallclock time to indicate that an event has not occurred "until now."[4] Thus, in practice, non-occurrences primarily serve to associate wallclock time time—an implicit concept—with event occurrence or non-occurrence. This suggests that there may be a way to keep non-occurrences implicit, thereby saving the space overhead required for storing and garbage-collecting the non-occurrences.

There is another problem with the non-blocking explicit representation of non-occurrences: detecting an event occurrence requires polling the occurrence stream continually. Since polling can only happen at discrete intervals, this introduces a latency between event determination and reaction. Unfortunately, for simple controller-model-view interactions, where activation of a control is supposed to produce an immediate change in the model and the view, even short delays of 1/20 of a second are quite visible to the user. Moreover, polling can be computationally expensive. Hence, the streams representation of events is not synchronous.

12.5.4 Determination vs. Occurrence

The key to having an implicit representation of event non-occurrence is the difference between the time at which an event occurrence is known and the time it actually happens. While the latter is the *time of occurrence*, the other is the *time of determination*.

Consider an "alarm event" constructed by the alarm procedure `alarm` in Lula's reactivity subsystem:

`(alarm` *time*`)` procedure

`Alarm` returns an event with exactly one occurrence at time *time*. If `alarm` is called with a time in the future like

```
(define e (alarm (+ (current-time) 5.0)))
```

at time t, the time of occurrence of `e` is very close to $t + 5$, whereas the time of determination is t. For practical purposes, the time of determination cannot be significantly later than the time of occurrence: to be accurate, the sampling engine needs to know about event occurrences which happened in the past as soon as possible.

Conversely, this means the sampling engine might just as well go ahead and assume that, when it is sampling a reactive value at wallclock time, that all event occurrences which have not been determined will occur in the future. This leads to a simple strategy for detecting event non-occurrences even when they have no

[4] Actually, in Fran, event filters also generate non-occurrences, but it is not difficult to re-implement them without using non-occurrences.

explicit representation: attempt to sample the event for the next occurrence. There are three possible scenarios:

1. The sampling yields an occurrence in the past which has been determined in the past: occurrence.

2. The sampling yields an occurrence in the future which has been determined in the past: non-occurrence.

3. The sampling engine detects that sampling would block because no event occurrence is available. The determination time of the next occurrence is in the future, so it is safe that the next occurrence time will also be in the future.

Consequently, sampling requires test which checks if sampling an event would block. The test itself must not block.

12.5.5 A Synchronous Implementation of Events

Lula's representation of events is based on the same idea as the stream-based representation with two notable differences: the representation of event non-occurrence is implicit, and the central data structure is no longer a promise but instead a *placeholder*.

Placeholders Placeholders abstract over the functionality of both promises and write-once variables, and are thread-safe.[5] This dual functionality makes them suitable both for "pre-fabricated" internal events such as the ones produced by `alarm` and interactive events hooked up to user interface controls.

Placeholders representing write-once variables result from a call to the `make-placeholder` constructor without any arguments:

`(make-placeholder)` procedure

An attempt to obtain its value will block until a call to `placeholder-set!`:

`(placeholder-set!` *placeholder obj*`)` procedure

The `placeholder-value` procedure returns the value of the placeholder if it has been set previously via `placeholder-set!`. If no call to `placeholder-set!` preceded the call of `placeholder-value`, `placeholder-value` will block until it occurs, returning the newly set value.

Another kind of placeholder contains a piece of code specified upon construction which `placeholder-value` calls when asked to obtain its value:

`(make-placeholder` *thunk*`)` procedure

Thunk is a procedure of no arguments. When the program calls `placeholder-value`, it will evaluate the thunk, memoize its value and return it. Evaluating the thunk may block.

A third form of calling `make-placeholder` allows the caller to be more specific as to the blocking behavior of `placeholder-value`:

`(make-placeholder` *thunk blocking-info*`)` procedure

[5] Placeholders started out as an implementation of the communication mechanism of the same name in Kali Scheme [Cejtin *et al.*, 1995] and newer versions of Scheme 48 [Kelsey and Rees, 1995], but has evolved beyond the original idea. Lula's placeholders subsume the functionality present in these systems.

In its simplest form, *blocking-info* is a boolean value, in which case it specifies whether evaluating *thunk* would block. It may also be another placeholder which signals that evaluating *thunk* will also attempt to obtain the value of that placeholder. Finally, *blocking-info* can also be a thunk whose value, when evaluated, will tell whether evaluating *thunk* would block.

Another selector beside `placeholder-value`, `placeholder-ready?`, tells if calling `placeholder-value` on the placeholder would return immediately or block:

`(placeholder-ready?` *placeholder*`)` procedure

Events as Placeholder Lists Lula's reactivity subsystem represents events as two lazy lists represented using placeholders: the first list, `head`, starts with the the first event occurrence. The second list, `current`, is for interactive events, and points to a placeholder representing the first undetermined event occurrence:

```
(define-record-type :event
  (real-make-event head current)
  event?
  (head event-head)
  (current event-current set-event-current!))
```

An event starts off with both lists being the same:

```
(define (make-event . args)
  (if (not (null? args))
      (make-event (car args))
      (let ((plist (make-placeholder)))
        (make-event plist plist))))
```

For an interactive event, a callback from a user interface control will typically write to the `current` placeholder and advance it by one. The callback must supply the occurrence data:

```
(define (determine-event! event occurrence)
  (let ((next (make-placeholder)))
    (placeholder-set! (event-current event)
                      (cons occurrence next))
    (set-event-current! event next)))
```

Note that the obvious way of creating and feeding interactive events is usually not a good idea. A programmer might be tempted to define an event and then repeatedly call `determine-event!` on it:

```
(define e (make-event))
...
   (determine-event! e (make-occurrence (current-time) (delay 'foo)))
```

This program will not allow the garbage collector to reclaim `e`. Consequently, `e` will accumulate ever more occurrences which take up an increasing amount of space: the provider of `e` effectively keeps `e` alive, whereas really the client of `e`—ultimately the sampling engine—should determine how far back occurrences of `e` should be remembered. Hence, the provider, after determining an event occurrence, should replace the event by another which does not hold on to that occurrence. The `ping-event` procedure returns the desired event:

```
(define (ping-event event thing)
  (determine-event! event (make-occurrence (current-time)
                                           (delay thing)))
  (rest-event event))
```

The `rest-event` procedure skips the first occurrence—in the case of `ping-event`, it is the occurrence just generated:

```
(define (rest-event event)
  (make-event (cdr (placeholder-value (event-head event)))))
```

With `ping-event`, the code above turns into the following space-safe variant:

```
(define e (make-event))
...
   (set! e (ping-event e 'foo))
```

On the other hand, pre-determined events provide the occurrence list to `make-event` and use the other kind of placeholder. `Alarm` has only one occurrence; obtaining it can never block:

```
(define (alarm time)
  (make-event
    (make-placeholder (lambda ()
                        (cons
                          (make-occurrence time (tdelay 'alarm))
                          (make-placeholder (lambda () '()) #f)))
                      #f)))
```

The procedure implementing the semantics of events, `occurrences`, utilizes the implicit representation of non-occurrence: given a sampling time, `occurrences` assumes that, if the event has not been determined yet, it will not occur before the sampling time. Hence, in first case of the `cond` of the loop, `occurrences` just sticks `#f` into the output list if the placeholder is not ready yet. `Occurrences` returns a "traditional" even-style lazy list:

```
(define (occurrences event time-stream)
  (delay
    (let loop ((head (event-head event))
               (time-stream time-stream))
      (cond
       ((not (placeholder-ready? head))
        (cons #f
              (delay (loop head (stream-cdr time-stream)))))
       ((null? (placeholder-value head))
        (force (repeat-stream #f)))
       (else
        (let* ((pair (placeholder-value head))
               (occurrence (car pair))
               (next-time (stream-car time-stream)))
          (if (<= next-time (occurrence-time occurrence))
              (cons #f
                    (delay (loop head
                                 (stream-cdr time-stream))))
              (cons occurrence
                    (delay (loop (cdr pair)
                                 (stream-cdr time-stream))))))))))))
```

It is important that combinators constructing events from other events propagate the blocking behavior. Many such combinators simply match occurrence with occurrence; hence, obtaining an occurrence of the resulting event will incur obtaining

a corresponding occurrence of the original. Consider the `handle-event` procedure, an implementation of the handle-event operator introduced in Section 12.2.2. `Handle-event` chains the placeholder of the resulting occurrence list to the corresponding placeholder of its argument:

```
(define (handle-event event proc)
  (make-event
   (let loop ((head (event-head event)))
     (make-placeholder
      (lambda ()
        (let ((pair (placeholder-value head)))
          (if (null? pair)
              '()
              (let* ((rest (cdr pair))
                     (occurrence (car pair))
                     (otime (occurrence-time occurrence))
                     (promise (occurrence-promise occurrence)))
                (cons (make-occurrence otime
                                       (delay
                                        (proc otime
                                              (force promise)
                                              (make-event rest))))
                      (loop rest))))))
      head))))
```

The code for `sample-event` is still straightforward, and very similar to the implementation in a direct stream-based implementation of events. Still, there are a few operators which require more effort. An example for a more troublesome operation is `filter-map-event`:[6]

`(filter-map-event` *event proc*`)` procedure

Proc is a procedure which accepts one argument of the same type as the event occurrence values. *Proc* returns either `#f` or some other value. `Filter-map-value` will produce an event with a subset of the occurrences of its argument; it omits occurrences for which *proc* returns `#f`, and replaces the occurrence values of the others by the return value of *proc*, respectively.

How is `filter-map-event` to determine if obtaining its first occurrence would block? Potentially, *proc always* returns `#f`, and thus obtaining the value could start a computation which takes an infinite amount of time. One way to solve the problem would be to use a thread that speculatively samples *event* and uses timeouts to always keep the new event up-to-date. However, it seems that the problem of infinite computation can only occur in degenerate situations. Two conditions have to coincide:

1. *Proc* returns `#f` for a large prefix of the event occurrences of *event*.
2. The placeholders for this large prefix do not block.

Hence, the problem is mostly relevant with pre-determined events with many non-blocking occurrences in a row. Such events are rare (a periodic alarm comes to mind), and applying a filter to an artificially generated event seems to make little sense in general. Thus, an implementation of `filter-map-event` which does not require spawning any threads is possible. It merely requires some care in specifying the blocking behavior of its occurrences. The generation of the resulting event

[6] I owe Peter Biber for spotting the problem.

occurrences is straightforward—a loop looks at the event occurrences, constructing new occurrences from *proc* matches, and skipping non-matches:

```
(define (filter-map-event event proc)
  (make-event
   (let loop ((head (event-head proc)))
     (make-placeholder
      (lambda ()
        (let inner-loop ((head head))
          (let ((pair (placeholder-value head)))
            (if (null? pair)
                '()
                (let* ((occurrence (car pair))
                       (stuff (proc (force (occurrence-promise occurrence)))))
                  (if stuff
                      (cons (make-occurrence (occurrence-time occurrence)
                                             stuff)
                            (loop (cdr pair)))
                      (inner-loop (cdr pair))))))))))))
```

`Filter-map` actually exploits `make-placeholder`'s ability to accept a thunk specifying the blocking behavior of the placeholder. If obtaining the next occurrence of *event* would block, so would obtaining the next occurrence of the filtered event:

```
      (lambda ()
        (let loop ((head (event-head))
          (if (not (placeholder-ready? head))
              #t
```

If *event* has no more occurrences (and if obtaining that fact would not block), neither has the filtered event:

```
              (let* ((pair (placeholder-value head)))
                (if (null? pair)
                    #f
```

Finally, if the next occurrence matches, the filtered event will also have a corresponding occurrence—obtaining it will not block:

```
                    (let* ((occurrence (car pair))
                           (stuff
                            (proc (force (occurrence-promise occurrence)))))
                      (if stuff
                          #f
                          (loop (cdr pair))))))))))))))
```

Should the restrictions `filter-map-event` imposes on its argument event become relevant, it is easy enough to convert an event into an equivalent *synchronous event*, where event determination always happens after occurrence, preferably very close to it. To achieve this effect, the `synchronous-event` procedure maps over the event occurrences, delaying the availability of each occurrence until its occurrence time has come.

```
(define (synchronous-event event)
  (make-event
   (let loop ((head (event-head event)))
     (make-placeholder
```

```
      (lambda ()
        (let ((pair (placeholder-value head)))
          (if (null? pair)
              '()
              (let* ((occurrence (car pair))
                     (delay-time (- (occurrence-time occurrence)
                                    (current-time))))
                (if (positive? delay-time)
                    (sleep delay-time))
                (cons occurrence
                      (loop (cdr pair)))))))
      (lambda ()
        (if (not (placeholder-ready? head))
            #t
            (let ((pair (placeholder-value head)))
              (if (null? pair)
                  #f
                  (let* ((occurrence (car pair))
                         (delay-time (- (occurrence-time occurrence)
                                        (current-time))))
                    (positive? delay-time)))))))))
```

With the synchronous representation, the operation most difficult to implement is `merge`: `merge` accepts two events as arguments and returns an event with the occurrences of both. In principle, `merge` performs a standard list-merge operation on the occurrence lists. However, with the synchronous representation, `merge` must preserve synchronousness, even when only one or none of the two events have been determined at sampling time. In case both events have occurrences determined, the code is straightforward:

```
(define (merge event-1 event-2)
  (make-event
   (let loop ((head-1 (event-head event-1))
              (head-2 (event-head event-2)))
     (make-placeholder
      (lambda ()
        (let ((ready-1? (placeholder-ready? head-1))
              (ready-2? (placeholder-ready? head-2)))
          (cond
           ((and ready-1? ready-2?)
            (let ((pair-1 (placeholder-value head-1))
                  (pair-2 (placeholder-value head-2)))
              (cond
               ((null? pair-1) pair-2)
               ((null? pair-2) pair-1)
               (else
                (let ((occurrence-1 (car pair-1))
                      (occurrence-2 (car pair-2)))
                  (let ((otime-1 (occurrence-time occurrence-1))
                        (otime-2 (occurrence-time occurrence-2)))
                    (if (<= otime-1 otime-2)
                        (cons occurrence-1
                              (loop (cdr pair-1) head-2))
                        (cons occurrence-2
                              (loop head-1 (cdr pair-2)))))))))))
```

If only `event-1` has a determined occurrence, `merge` may need to wait until either the occurrence time of that occurrence or the determination time of the next occurrence of `event-2`, whichever comes first:

```
(ready-1?
 (let ((pair-1 (placeholder-value head-1)))
   (if (null? pair-1)
       (placeholder-value head-2)
       (let* ((occurrence-1 (car pair-1))
              (otime-1 (occurrence-time occurrence-1))
              (delay-time (- otime-1 (current-time))))
         (if (positive? delay-time)
             (begin
               (placeholder-wait/timeout head-2 delay-time)
               (placeholder-value (loop head-1 head-2)))
             (cons occurrence-1
                   (loop (cdr pair-1) head-2)))))))
```

The `placeholder-wait/timeout` procedure waits for either the placeholder value to become available or for a timeout to expire. The case where `event-2` has a determination occurrence is analogous:

```
(ready-2?
 (let ((pair-2 (placeholder-value head-2)))
   (if (null? pair-2)
       (placeholder-value head-2)
       (let* ((occurrence-2 (car pair-2))
              (otime-2 (occurrence-time occurrence-2))
              (delay-time (- otime-2 (current-time))))
         (if (positive? delay-time)
             (begin
               (placeholder-wait/timeout head-1 delay-time)
               (placeholder-value (loop head-1 head-2)))
             (cons occurrence-2
                   (loop head-1 (cdr pair-2)))))))))
```

Finally, if neither `event-1` nor `event-2` has a determined occurrence available, `merge` needs to wait for one; `placeholder-wait-2` waits until either of its argument placeholders produces a value:

```
(else
 (placeholder-wait-2 head-1 head-2)
 (placeholder-value (loop head-1 head-2))))))
```

The code for computing the blocking behavior is analogous:

```
(lambda ()
 (let ((ready-1? (placeholder-ready? head-1))
       (ready-2? (placeholder-ready? head-2)))
   (cond
    ((and ready-1? ready-2?) #f)
    (ready-1?
     (let ((pair-1 (placeholder-value head-1)))
       (if (null? pair-1)
           #t
           (let* ((occurrence-1 (car pair-1))
```

```
                (otime-1 (occurrence-time occurrence-1))
                (delay-time (- otime-1 (current-time))))
          (positive? delay-time)))))
(ready-2?
 (let ((pair-2 (placeholder-value head-2)))
   (if (null? pair-2)
       #t
       (let* ((occurrence-2 (car pair-2))
              (otime-2 (occurrence-time occurrence-2))
              (delay-time (- otime-2 (current-time))))
         (positive? delay-time)))))
(else
 #t))))))))
```

12.5.6 General Behaviors

Specific application domains of FRP—say, sprite-based animation, robotics, or, in this case, lighting—can benefit from a specialized representation for reactive values, specifically for behaviors. Specialized representations communicate more domain-specific structural information. Often, this information, together with structural domain knowledge, allow the application to exploit simplification and optimization opportunities not possible with the "plain" behaviors provided by Lula's reactivity subsystem. For example, Fran uses a representation for 2D image animations amenable to translation into the sprite representation of Microsoft's DirectX engine [Elliott, 1999, Elliott, 1998a].

To avoid excessive code duplication, it is desirable to factor the representation of behaviors into a small "core" which actually accesses the representation, and a layer of high-level combinators which do not. In the case of behaviors, the core reduces to four procedures. This list is identical to the one in Fran [Elliott, 1998a]:

`(until` *behavior event*`)` procedure

`Until` is the implementation of the until operator introduced in Section 12.2.3. The behavior `until` returns behaves like *behavior* until the first occurrence of *event* which must produce another behavior as its value; this is the new behavior from then on.

`(after-times` *behavior time-list*`)` procedure

`After-times` is the fundamental aging operator: it accepts a behavior and a lazy lists of times. It returns a lazy list of behaviors corresponding to the elements of *time-list*. Each behavior in the output list "forgets" all samples before its corresponding time in *time-list*, thereby potentially releasing memory previous elements occupy. `After-times` is primarily for internal use by higher-level combinators.

`(time-transform` *behavior time-behavior*`)` procedure

`Time-transform` is the implementation of time transformation as described in Section 12.2.1: *time-behavior* supplies a new local time for *behavior*.

`(cond-unopt` *bool-behavior behavior-1 behavior-2*`)` procedure

`Cond-unopt` is a simple conditional: it produces behavior which assumes the value of *behavior-1* at times *bool-behavior* produces true, and *behavior-2* at times the *bool-behavior* produces false.

Fran uses Haskell type classes to abstract over all so-called *generalized behaviors* which implement these four operators. Lula uses a traditional data-directed

programming approach [Abelson *et al.*, 1996], employing MzScheme's object system [Flatt, 2000] for the implementation. An interface captures generalized behaviors:

```
(define gbehavior<%>
  (interface ()
    until
    after-times
    time-transform
    cond-unopt))
```

(The trailing `<%>` is merely a naming convention.) This approach covers almost all the grounds that Fran's type-class-based implementation does; the only thing it does not provide is parameterized lifting, such as from pairs of behaviors to behaviors of pairs. The procedures specified above merely call the methods of the interface:

```
(define (until gbehavior event)
  (send gbehavior until event))

(define (after-times gbehavior time-stream)
  (send gbehavior after-times time-stream))

(define (time-transform gbehavior time-behavior)
  (send gbehavior time-transform time-behavior))

(define (cond-unopt bool-behavior behavior-1 behavior-2 time-behavior)
  (send behavior-1 cond-unopt bool-behavior behavior-2))
```

12.5.7 Recursive Reactive Values Revisited

A number of behaviors arising in applications are naturally recursive. The most prominent example is the integral, which also generally is a good example of FRP. Typical approximation algorithms add up pieces of the area under a function graph. The integral computation adds every new piece to the area computed so far. Lula's implementation of integrals uses Euler's algorithm. A naive programmer might produce the following first shot, using an event `step-event` to provide the approximation intervals:

```
(define (integral-1 behavior step-event)
  (letrec ((new-sample
            (map-event
             (snapshot (time-only-event step-event)
                       (!cons behavior integral-behavior))
             (lambda (stuff)
               (let ((t0 (car stuff))
                     (y (car (cdr stuff)))
                     (int-so-far (cdr (cdr stuff))))
                 (!+ (! int-so-far)
                     (!* (!- time (! t0))
                         (! y)))))))))
    (integral-behavior (switcher (! 0) new-sample)))
  integral-behavior))
```

`Integral-1` creates two mutually recursive reactive values: `new-sample` is an event providing a new integral value for each occurrence, sampling `behavior` every time. `Integral-behavior` is the corresponding behavior.

`Integral-1` calls a number of procedures whose names start with `!`. These are all lifted versions of the corresponding procedures without a `!`. The `!` specifically creates a constant behavior from a value. `!+` takes two behaviors of numbers as arguments and returns a behavior of sums. Analogously, `!-` is the lifted version of `-`, `!*` is the lifted version of `*` and `!cons` is the lifted version of `!cons`.

Furthermore, `time-only-event` turns any event whose occurrences have their timestamps as values. Hence, the call to `snapshot` returns an event that samples three values:

- its timestamp,
- the current value of the behavior to be integrated, and
- the integral approximation accumulated so far.

`Map-event` turns these three values into a behavior of integral approximations from the event occurrence on.

The `switcher` procedure assembles the piecewise approximation behaviors into one, starting with a constant 0 up to the first occurrence of the sampling event.

Now, even though the logic underlying the above implementation is correct, it will not work: Scheme is a strict language, and the both right-hand-sides of the bindings evaluate the other binding respectively. Thus, a correct implementation of `integral-1` must delay both `new-sample` and `integral-behavior`. Unfortunately, this changes the representation from a behavior or an event to a promise respectively, and the reactivity subsystem cannot deal with those.[7]. Fortunately, the subsystem does use promises for the representation of behaviors, and a simple procedure `promise->behavior` converts a promise of a behavior into a behavior without forcing the promise:

```
(define (promise->behavior promise)
  (make-behavior
   (delay
     (force
      (behavior-*-promise (force promise))))))
```

The correct implementation of `integral-1` uses the `promise->behavior` procedure to delay `integral-behavior`; and ordinary promise suffices for `new-sample`:

```
(define (integral-1 behavior step-event)
  (letrec ((new-sample
            (delay
              (map-event
               (snapshot (time-only-event step-event)
                         (!cons behavior integral-behavior))
               (lambda (stuff)
                 (let ((t0 (car stuff))
                       (y (car (cdr stuff)))
                       (int-so-far (cdr (cdr stuff))))
                   (!+ (! int-so-far)
                       (!* (!- the-time (! t0))
                           (! y))))))))
           (integral-behavior
            (promise->behavior
             (delay
              (switcher (! 0) (force new-sample))))))
    integral-behavior))
```

[7] Here, Haskell makes the programmer's life much easier by delaying everything implicitly.

Chapter 13

Functional Reactive Lighting

I'm always ready to dance. But I need me a kiss first, honey. Just one?
—Lula in *Wild at Heart*

Functional Reactive Programming has direct application to animated lighting. Lula internally expresses all light changes as animations in term of FRP. The key ingredient for the representation of light changes is the *preset behavior*, a specialized representation for behaviors over presets. Lula offers simplified graphical user interface components for simple light changes and effects. For more complex animation, the user has direct access to FRP via a built-in domain-specific programming language called *Lulal*, a restricted dialect of Scheme with static typing. Lulal allows the user to assemble preset behaviors into *tasks* which allow the sequencing of animations.

This chapter introduces the concepts and machinery behind Lula's substrate for animated lighting. It starts off with a formulation of the most important goals of a lighting animation engine from the application domain. It then goes on with a formal description of Lulal, and shows how to achieve the goals stated earlier. The chapter concludes with an overview of the implementation issues; the concern the implementation of Lulal itself, and the sampling engine which turns Lulal expressions into lighting actions.

13.1 Sanity Checks

Here are some sample jobs from actual show designs the reactive subsystem of Lula must be able to perform:

- The lighting intensity "breathes," dimming and brightening periodically according to a sine curve.
- A moving light describes a circle on the floor of the stage around a specified center, with a specified radius.
- A moving light describes a circle whose center and radius are under interactive control.
- The lighting changes color when a dancer crosses a light barrier on stage.
- 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.
- Several lights simulate a flickering fire when activated at random.

Section 13.6 shows how Lulal copes with these examples.

⟨expression⟩ ::= ⟨variable⟩
| ⟨literal⟩
| ⟨procedure call⟩
| ⟨lambda expression⟩
| ⟨conditional⟩
| ⟨let expression⟩
| ⟨values expression⟩
| ⟨random expression⟩
| ⟨derived expression⟩
⟨literal⟩ ::= ⟨boolean⟩ | ⟨number⟩ | ⟨cue name⟩
⟨cue name⟩ ::= ⟨string⟩
⟨procedure⟩ ::= `(`⟨operator⟩ ⟨operand⟩`)`
⟨operator⟩ ::= ⟨expression⟩
⟨operand⟩ ::= ⟨expression⟩
⟨lambda expression⟩ ::= `(lambda` ⟨formals⟩ ⟨body⟩`)`
⟨formals⟩ ::= `(`⟨variable⟩`*)`
⟨body⟩ ::= ⟨expression⟩
⟨conditional⟩ ::= `(`⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`
⟨test⟩ ::= ⟨expression⟩
⟨consequent⟩ ::= ⟨expression⟩
⟨alternate⟩ ::= ⟨expression⟩
⟨let expression⟩ ::= `(let (`⟨binding spec⟩`*)` ⟨body⟩`)`
⟨binding spec⟩ ::= `(`⟨variable⟩ ⟨expression⟩`)`
⟨values expression⟩ ::= `(values` ⟨expression⟩$^+$`)`
⟨random expression⟩ ::= `(random` ⟨expression⟩$^+$`)`

Figure 13.1: Lulal primitive expression syntax.

13.2 The Lulal Language

Lulal is a higher-order, purely functional, strongly typed language with parametric polymorphism. Its syntax is mostly borrowed from Scheme [Kelsey *et al.*, 1998]. 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:

- Explicit recursion is not allowed.
- Lulal is strongly typed. Its type system is a modified version of the popular Hindley/Milner system [Damas and Milner, 1982, Hindley, 1969].
- The language allows a limited form of overloading of constant values with constant behaviors.

13.2.1 Lulal Syntax

Figure 13.1 shows Lulal's expression syntax. The grammar is basically an excerpt of the Scheme grammar in R^{5}RS [Kelsey *et al.*, 1998]. It builds on the same lexical structure as Scheme whose grammar was omitted here for brevity. As the grammar

⟨derived expression⟩ ::= ⟨sequence expression⟩
| ⟨let* expression⟩
| ⟨and expression⟩
| ⟨or expression⟩
| ⟨cond expression⟩
⟨let* expression⟩ ::= `(let*` (⟨binding spec⟩*) ⟨body⟩`)`
⟨sequence expression⟩ ::= `(sequence` ⟨sequence step⟩$^+$`)`
⟨sequence step⟩ ::= ⟨expression⟩
| (⟨expression⟩ `=>` ⟨variable⟩`)`
⟨and expression⟩ ::= `(and` ⟨test⟩*`)`
⟨or expression⟩ ::= `(or` ⟨test⟩*`)`
⟨cond expression⟩ ::= `(cond` ⟨cond clause⟩* `(else` ⟨expression⟩`))`
⟨cond clause⟩ ::= (⟨test⟩ ⟨expression⟩`)`

Figure 13.2: Lulal derived expression syntax.

⟨program⟩ ::= ⟨definition⟩*
⟨definition⟩ ::= `(define` ⟨variable⟩ ⟨expression⟩`)`
| `(define` (⟨variable⟩ ⟨definition formals⟩`)` ⟨body⟩`)`
⟨definition formals⟩ ::= ⟨variable⟩*

Figure 13.3: Lulal program syntax.

shows, some derived forms are missing—everything having to do with assignment and side effects, `case`, `letrec`, `do`, and `delay`. Also, Lulal has neither quote nor backquote syntax. `Lambda` is restricted to a fixed-length parameter list.

Strings make little sense in Lula, so their syntax is available for a different purpose: a string literal stands for the name of a cue. Part of the syntactic analysis is checking that all cues named in a Lulal program actually exist.

Lulal has two primitive expression types not in the Scheme grammar. `Values` is similar to the primitive procedure of the same name in Scheme: it constructs a tuple of its arguments. It is in the grammar rather than the list of primitive procedures because its type would not be expressible in the Hindley/Milner system. A `random` form evaluates to one of its operands, chosen at random during runtime. It also is in the syntax rather than the library for typing reasons.

Another small difference to the Scheme grammar is the listing of `let` under the primitive expression syntax rather than the derived one. This is also for typing reasons; Hindley/Milner depends on `let` being a primitive form.

Figure 13.2 shows the syntax of the derived expressions in Lulal. They are a fairly standard subset of the corresponding part of the Scheme syntax with one exception: `sequence`. `Sequence` is a special syntactic construct for the sequencing in the task monad, akin to Haskell's `do` syntax [Haskell98, 1998]. Its precise semantics is explained in Section 13.5.

A Lulal program is a sequence of definitions. Figure 13.3 shows the definition syntax which again is a subset of the Scheme syntax. A Lula show can then use tasks defined in the Lulal program for defining events (see Chapter 7).

13.2.2 Lulal Semantics

$$
\begin{array}{rcll}
\tau & = & b & b \text{ base type} \\
 & | & v & v \text{ type variable} \\
 & | & \tau_1 \rightarrow \tau_2 & \\
 & | & \tau_1 \times \cdots \times \tau_n & \\
 & | & behavior_\tau & \\
 & | & event_\tau & \\
 & | & task_\tau &
\end{array}
$$

Figure 13.4: Lulal type language.

Figure 13.4 shows the type language of Lulal. The base types are unit, numbers, booleans, and (points in) time. In addition, the language includes function and tuple types. Moreover, behaviors, events, and tasks are primitive types.

Lulal handles tuples in a way akin to ML [Milner *et al.*, 1997]: conceptually, every procedure has exactly one parameter and one return value. A `lambda` with two or more formals in fact creates a procedure with a tuple-type argument, whose body implicitly binds the parameters to the tuple components. `Values` constructs tuples which are first-class values in Lulal. Most of the time, this distinction from Scheme is of little consequence for the Lulal programmer who knows Scheme. However, some unusual constructs are possible. The following expression yields 65, for instance:

```
((lambda (x y) (+ x y)) (values 23 42))
```

As in Scheme, `(values` x`)` for any expression x is equivalent to x itself: a one-component tuples is identified with its component.

13.2.3 Behaviors and Events

Lulal's value domain includes parameterized behaviors, events, and tasks for constructing reactive values.

The semantics of behaviors and events is that defined by the primitives of Lula's reactive subsystem as explained in the previous chapter and detailed below.

13.2.4 Tasks

Tasks represent actions to be done in the reactive framework: a task defines a lighting change behavior as well as a time at which it ends. When the action of a task is done, the task also produces an exit value.

Tasks form a *monad* [Moggi, 1988, Wadler, 1995], 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 [Peterson and Hager, 1999]. Lulal supports the usual monadic combinators `return` and `bind`. 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.

A more thorough explanation of the way tasks work along with the primitives available for them in Lulal is below in Section 13.5.

13.2.5 Type Inference and Overloading

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 is most useful for numeric literals. Consider the following example:

```
(sin (+ (seconds time) 1.0))
```

Lulal automatically coerces the value of the literal `1.0` to a constant behavior.

`If` is also thus overloaded, and doubles as a conditional over behaviors. The same holds true for the derived forms `cond`, `and`, and `or`.

13.3 Built-In Bindings

This section gives a brief overview of the primitive bindings available to Lulal programs. Many of these procedures correspond directly to primitives in Lula's reactivity subsystem. For details, refer to the previous chapter.

13.3.1 Reals

All standard numerical operators are available in versions lifted to behaviors: `+`, `-`, `*`, `/`, `sin`, `asin`, `cos`, `acos`, `tan`, `atan`, `exp`, `expt`, `sqrt`, and `abs`, along with predicates `=`, `<`, `>`, `<=`, `>=`.

Lulal has two notable additions specific to behaviors:

`integral:` $behavior_{real} \rightarrow behavior_{real}$
`(integral` *behavior*`)` procedure
`derivative:` $behavior_{real} \rightarrow behavior_{real}$
`(derivative` *behavior*`)` procedure

These procedures compute numerical approximations to the integral or the derivative of the argument behavior, respectively. The sampling interval is determined by a heartbeat internal to Lula, at most as long as the central sampling interval.

13.3.2 Time

`time:` $behavior_{time}$
`time` value

This is the wallclock time behavior.

`seconds:` $behavior_{time} \rightarrow behavior_{real}$
`(seconds` *time*`)` procedure

This procedure converts a time behavior into a behavior over reals. The reals returned are measured in seconds.

`later:` $time \times real \rightarrow time$
`(later` *time delay*`)` procedure

This procedure adds an offset to a point in time.

`delayed:` $behavior_{time} \times behavior_{real} \rightarrow behavior_{time}$
`(delayed` *time delay*`)` procedure

This procedure delays a behavior of points in time by *delay* seconds.

`slower:` $behavior_{time} \times real \rightarrow behavior_{time}$
`(slower` *time factor*`)` procedure
`faster:` $behavior_{time} \times real \rightarrow behavior_{time}$
`(faster` *time factor*`)` procedure

These procedure slow down (or speed up, respectively) a behavior of points in time by a factor of *factor*.

`time-transform`: $behavior_{\alpha} \times behavior_{time} \rightarrow behavior_{\alpha}$
`(time-transform` *behavior time-behavior*`)`

`Time-transform` creates a time-transformed version of *behavior* by feeding it the points in time from *time-behavior*.

13.3.3 Pan/Tilt

These behaviors specify transformations of the pan/tilt parameter.

`pan/tilt`: $behavior_{real} \times behavior_{real} \rightarrow behavior_{pan/tilt}$
`(pan/tilt` *pan tilt*`)` procedure

`Pan/tilt` creates an absolute pan/tilt transformation behavior from separate behaviors *pan* and *tilt*. *Pan* and *tilt* are both in radians.

`xyz`: $behavior_{real} \times behavior_{real} \times behavior_{real} \rightarrow behavior_{pan/tilt}$
`(xyz` *x y z*`)` procedure

`Xyz` creates an absolute pan/tilt transformation behavior from separate behaviors for 3D Cartesian coordinates. Lula converts these coordinates internally to pan/tilt settings with the help of the calibration provided by the operator.

`xyz-offset`: $behavior_{real} \times behavior_{real} \times behavior_{real} \rightarrow behavior_{pan/tilt}$
`(xyz-offset` *x-offset y-offset z*`)` procedure

`Xyz-offset` creates an absolute pan/tilt transformation behavior from separate behaviors for 3D Cartesian coordinates. `Xyz-offset` works in a somewhat subtle way: the *x-offset* and *y-offset* arguments specify offsets in the horizontal plane at Z coordinate *z* (see Section 10.6).

13.3.4 Colors

`lee`: $behavior_{integer} \rightarrow behavior_{color}$
`(lee` *n*`)` procedure
`rosco`: $behavior_{integer} \rightarrow behavior_{color}$
`(rosco` *n*`)` procedure

These procedures create color behaviors, mapping from the familiar identification numbers from Lee [LEE Filters, 2000] and Rosco [Rosco Laboratories, 2000].

`rgb`: $behavior_{real} \times behavior_{real} \times behavior_{real} \rightarrow behavior_{real}$
`(rgb` *r g b*`)` procedure
`hsv`: $behavior_{real} \times behavior_{real} \times behavior_{real} \rightarrow behavior_{real}$
`(hsv` *h s v*`)` procedure
`cmy`: $behavior_{real} \times behavior_{real} \times behavior_{real} \rightarrow behavior_{real}$
`(cmy` *c m y*`)` procedure

These procedures construct color behaviors according to the Red/Green/Blue, Hue/Saturation/Value, and Cyan/Magenta/Yellow color models. (For details about the various color models, refer to the literature [Foley *et al.*, 1996].)

13.3.5 Preset Behaviors

Presets are parameter settings for fixtures (see Section 11.14 for details). 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:` $preset\text{-}behavior \times preset\text{-}behavior \rightarrow preset - behavior$
`(restrict-with` $preset_1$ $preset_2$`)` procedure
`subtract:` $preset\text{-}behavior \times preset\text{-}behavior \rightarrow preset - behavior$
`(subtract` *preset* *preset*`)` procedure

These are the restriction and difference operators from the cue algebra (see Chapter 11). HTP is not available because it may lead to conflicts during the sampling of a Lulal-defined behavior.

`scale:` $behavior_{real} \times preset\text{-}behavior \rightarrow preset - behavior$
`(scale:` *real* *preset*`)` procedure

This procedure creates a scaled preset behavior from a real behavior defining a scale factor and a preset behavior.

`with-color:` $behavior_{color} \times preset\text{-}behavior \rightarrow behavior_{color}$
`(with-color` *color* *preset*`)` procedure

`With-color` creates a colored version of a preset behavior. The *color* argument must be a color behavior constructed with the primitives described in Section 13.3.4.

`with-pan/tilt:` $behavior_{pan/tilt} \times preset\text{-}behavior \rightarrow behavior_{pan/tilt}$
`(with-pan/tilt` *pan/tilt* *preset*`)` procedure

`With-pan/tilt` creates a version of a preset behavior with all moving lights of the preset at the pan/tilt values specified by the *pan/tilt* argument. The *pan/tilt* argument must be a pan/tilt behavior constructed with the primitives described in Section 13.3.3.

13.4 Events

This section describes the means for event construction available in Lulal. These correspond mostly to similarly-named primitives in the reactivity library:

`never:` $event_{unit}$
`alarm:` $time \rightarrow event_{unit}$
`(alarm` *time*`)` procedure
`periodic-alarm:` $time \times real \rightarrow event_{unit}$
`(periodic-alarm` *time* *period*`)` procedure

These all define primitive events: `never` has no occurrences ever, `alarm` has exactly one at the specified time, `periodic-alarm` an initial occurrence at *time*, then periodically after intervals *period* seconds long.

`map-event:` $event_\alpha \times (\alpha \rightarrow \beta) \rightarrow event_\beta$
`(map-event` *event* *proc*`)` procedure
`subst-event:` $event_\alpha \times \beta \rightarrow event_\beta$
`(subst-event` *event* *val*`)` procedure

`residual-event:` $event_\alpha \to event_\alpha$
`(residual-event` *event*`)` procedure
`time-event:` $event_\alpha \to event_{time}$
`(time-event` *event*`)` procedure
`timestamp-event:` $event_\alpha \to event_{time \times \alpha}$
`(timestamp-event` *event*`)` procedure

These procedures all define means of event handling—transforming events into new ones with occurrences at the same time but with different values. `Map-event` maps a procedure over the values. `Subst-event` substitutes a constant value for all event values. `Residual-event` turns an event into one whose values are the respective residual events. `Time-event` transforms an event whose values are the occurrence times. `Timestamp-event` keeps the original event values, but tuples them with the occurrence times.

`switcher:` $behavior_\alpha \times event_{behavior_\alpha} \to behavior_\alpha$
`(switcher` *behavior event*`)` procedure
`stepper:` $\alpha \times event_{alpha} \to behavior_\alpha$
`(stepper` *val event*`)` procedure

These two procedures turn events into behaviors whose value changes at each event occurrence. `Switcher` starts off with initial behavior *behavior* and then continues at each event occurrence with the behavior specified by the occurrence. `Stepper` assembles a piecewise-constant behavior from an initial value and new values specified by each event occurrence.

`merge:` $event_\alpha \times event_\alpha \to event_\alpha$
`(merge` $event_1$ $event_2$`)` procedure

This procedure merges the occurrences of two events into one.

13.5 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.

Intuitively, 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.

13.5.1 Basics

The primitive task construction procedure is `make-task`:

`make-task:` $((time \times preset) \to (behavior_{preset} \times event_\alpha)) \to task_\alpha$
`(make-task` *proc*`)` procedure

Proc is a procedure taking a start time and a starting preset as arguments and must return a preset behavior and an event whose first occurrence marks the end of the task's action.

`return:` $\alpha \to task_\alpha$
`(return` *val*`)` procedure

`Return` lifts a value into the task domain: it returns a task which terminates immediately, yielding the value *val*.

`bind:` $task_\alpha \times (\alpha \rightarrow task_\beta) \rightarrow task_\beta$
`(bind` *task proc*`)` procedure

`Bind` sequences two tasks. The action of the task the `bind` procedure returns first runs the action specified by *task*, feeds its result into *proc* and then runs the action of the task returned.

For convenient notation of `bind` cascades, Lulal provides the `sequence` construct. The operands of `sequence` are *steps*, each specifying a new application of `bind`. A step which is simply an expression throws away its exit value. A step of the form `(`*e* `=>` *v*`)` binds the exit value of *e* to *v* for the remaining steps. Here is a simple example:

```
(sequence
  (get-last-preset => preset)
  (x-fade 5 (restrict-with  preset "Sofa"))
  (x-fade 7 "Living Room"))
```

It translates to the following `bind` cascade:

```
(bind get-last-preset
      (lambda (preset)
        (bind (x-fade 5 (restrict-with preset "Sofa"))
              (lambda (dummy)q
                (x-fade 7 "Living Room")))))
```

In Scheme, `sequence` could be defined with the following macro:

```
(define-syntax sequence
  (syntax-rules (=>)
    ((sequence exp) exp)
    ((sequence (exp0 => v) step1 ...)
     (bind exp0
           (lambda (v)
             (sequence step1 ...))))
    ((sequence exp0 step1 step2 ...)
     (sequence (exp0 => v) step1 step2 ...))))
```

`get-start-time:` $task_{time}$
`get-start-time` value

`Get-start-time` is a task returning its start time immediately.

`get-last-preset:` $task_{preset}$
`get-last-preset` value

`Get-last-preset` is a task returning the terminating preset of the previous task immediately.

`wait:` $real \rightarrow task\,unit$
`(wait` *delay*`)` procedure

`Wait` creates a task which simply does nothing for *delay* seconds.

13.5.2 Combinators

`repeat:` $integer \times task_\alpha \to task_\alpha$
`(repeat` *n task*`)` procedure

The `repeat` combinator returns a task whose action executes the action defined by *task* *n* times in sequence.

`loop:` $task_\alpha \to task_\alpha$
`(loop` *task*`)` procedure

The action defined by the task returned by `loop` repeats the action of *task* endlessly.

`restrict-task-with:` $task_\alpha \times task_\alpha \to task_\alpha$
`(restrict-task-with` $task_1$ $task_2$`)` procedure

This procedure creates a parallel task: the task returned runs $task_1$ in parallel $task_2$. It terminates at the later of the termination times of the actions of $task_1$ and $task_2$. The preset behaviors defined by the tasks are defined by restriction of the two component behaviors.

13.5.3 Fades

`x-fade:` $real \times preset\text{-}behavior \to task_{unit}$
`(x-fade` *duration preset*`)` procedure

The `x-fade` procedure creates a task that performs a cross-fade to preset *preset* in *duration* seconds.

`in/out-fade:` $real \times real \times preset\text{-}behavior \to task_{unit}$
`(in/out-fade` *in out preset*`)` procedure

The `in/out-fade` procedure creates a task that performs an in/out fade to preset *preset* with in/out times *in* and *out*.

`in/out-delay-fade:` $real \times real \times real \times real \times preset\text{-}behavior \to task_{unit}$
`(in/out-delay-fade` *in-delay out-delay in out preset*`)` procedure

The `in/out-delay-fade` procedure creates a task that performs an in/out fade with delay to preset *preset* with in/out times *in* and *out* and delays *in-delay* and *out-delay*.

13.6 Simple Examples

Here are example programs to solve the sample assignments from Section 13.1 using Lulal. They all turn out to be very simple, but they give at least a glimpse of the expressive power of Lulal:

Breathing Light Here is a task which lets the lights defined by the cue `Living Room` breathe:

```
(make-task (lambda (start-time last-preset)
             (values
               (scale (sin (- time start-time)) "Living Room")
               never)))
```

Circle Assume `center` is a pair of pair of real behaviors representing the X and Y coordinates, respectively. Two procedures `get-x` and `get-y` are selectors for the such a tuple:

```
(define (get-x x y)
  x)

(define (get-y x y)
  y)
```

The `circle` procedure creates a pan/tilt transformation describing a circle at ground level:

```
(define (circle center radius)
  (xyz (+ (get-x center) (* radius (sin time)))
          (+ (get-y center) (* radius (cos time)))
          0))

(make-task (lambda (start-time last-preset)
             (values
               (with-pan/tilt (circle center radius) "Moving Light #1")
               never)))
```

Note that this pattern of a never-ending continuous task is easily turned into a procedure:

```
(define (preset-behavior->task behavior)
  (make-task (lambda (start-time last-preset)
               (values behavior never))))
```

Use of this procedure further simplifies the previous two examples.

Circle under interactive control For a circle with interactive controls, the user most define two sliders, one one-dimensional, the other two-dimensional. Assume two procedures `get-radius` and `get-center` return behaviors corresponding to these two sliders. Here is the task:

```
(let ((radius (get-radius-behavior))
      (center (get-center)))
  (preset-behavior->task
    (with-pan/tilt
      (circle center radius)
      "Moving Light #1")))
```

Reactive Color Change Assume `get-light-barrier-event` returns an event for the light barrier:

```
(with-color
  (stepper color-1
           (subst-event (get-light-barrier-event) color-2))
  "Light Cue")
```

Cascading Followspots Assume that some sensory equipment is hooked up to the console which reports the actor's position[1]. 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:

[1]Such systems are commercially available.

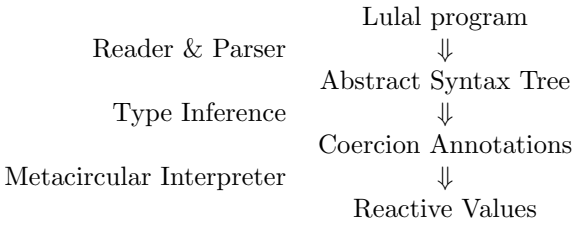

Figure 13.5: Lulal implementation stages.

```
(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")))))
```

Flickering Fire Assume the fire consists of five different, alternating cues called `Fire 1`, `Fire 2`, `Fire 3`, `Fire 4`, and `Fire 5`:

```
(define (some-fire)
  (random "Fire 1" "Fire 2" "Fire 3" "Fire 4" "Fire 5"))

(loop
  (sequence
    (x-fade 0 (some-fire))
    (wait 1)
    (x-fade 2 (some-fire))
    (x-fade 1 (some-fire))
    (x-fade 3 (some-fire))
    (wait 2)
    (x-fade 2 (some-fire))))
```

13.7 Implementation Notes

The implementation of Lulal is straightforward and follows established techniques for implementing domain-specific languages. Figure 13.5 shows the familiar stages a Lulal program goes through before it is ready for use by the main application.

13.7.1 Frontend Issues

The frontend parts of the Lulal implementation—the reader, the parser, and the type inference engine—depart from the traditional implementation folklore for Lisp-like languages which use `read` for parsing and perform no type checking:

Reader and parser build upon the *Zodiac* framework [Krishnamurthi, 1995] which is part of DrScheme: Zodiac first reads in the source program, performs macro expansion for the derived forms, and then produces an abstract syntax tree. This

syntax tree carries source location annotations. These annotations enable the syntax and type checker to provide the user with visual feedback about the location of errors, similar to the feedback DrScheme itself produces for its interactive development environment (see Chapter 7).

The type inference engine uses an implementation of the classic algorithm $\mathcal{W}$ [Damas and Milner, 1982]. The only significant departure is in the unification algorithm which handles the overloading. For every unification performed, the type inferencer returns coercions necessary to make the arguments type-compatible. The type inferencer returns an annotated abstract syntax tree with type and coercion information.

13.7.2 Implementing Tasks

The implementation of tasks is a specialized version of the task monads in the Frob system [Peterson and Hager, 1999]. The state it has to propagate is less elaborate than in Frob. Moreover, no exception handling is necessary since there is no feedback from the elecrical installation anyway.

The state propagated by the task monad is called a *task environment*. It carries the the start time and a reference to a *driver*, an internal data structure of the sampling subsystem described below in Section 13.7.4:

```
(define-record-type :task-environment
  (make-task-environment start-time driver)
  task-environment?
  (start-time task-environment-start-time)
  (driver task-environment-driver))
```

The task itself is represented by a procedure:

```
(define-record-type :task
  (make-task proc)
  task?
  (proc task-proc))
```

The `proc` component of a task of type $task_\alpha$ is a procedure of two arguments of type

$$\textit{task-environment} \times ((\textit{task-environment} \times \alpha) \to \textit{preset-behavior}) \to \textit{preset-behavior}$$

- The first parameter is a task environment.
- The second parameter is a *continuation* called when the present task terminates with the new environment and the new exit value.

The `return` and `bind` primitives are simple to implement:

```
(define (return k)
  (make-task (lambda (task-environment cont)
               (cont task-environment k))))
```

`Return` calls the continuation immediately, passing it the value injected into the task monad.

```
(define (bind task proc)
  (make-task
   (lambda (task-environment cont)
     ((task-proc task) task-environment
      (lambda (task-environment-1 result-1)
        ((task-proc (proc result-1)) task-environment-1 cont))))))
```

`Bind` creates a new task which first runs the action defined by `task`, passing it a continuation which will compute the task to follow after it, and applying that to the continuation.

A number of primitive task access the task environment.

```
(define get-task-environment
  (make-task (lambda (task-environment cont)
               (cont task-environment task-environment))))

(define get-start-time
  (bind get-task-environment
        (lambda (task-environment)
          (return (task-environment-start-time task-environment)))))

(define get-driver
  (bind get-task-environment
        (lambda (task-environment)
          (return (task-environment-driver task-environment)))))

(define get-last-preset
  (bind get-driver
        (lambda (driver)
          (driver-last-preset driver))))
```

At this point, suffice it to say that the `driver` gives access to the last preset of the previous action via the `driver-last-preset` selector.

Finally, `make-reactive-task` is the implementation of Lulal's `make-task` primitive described in Section 13.5.

```
(define (make-reactive-task proc)
  (make-task
   (lambda (task-environment cont)
     (call-with-values
      (lambda () (proc task-environment))
      (lambda (behavior end-event)
        (until
         behavior
         (map-event (timestamp-event end-event)
                    (lambda (stuff)
                      (let ((end-time (car stuff))
                            (end-event-value (cdr stuff)))
                        (cont (make-task-environment end-time
                                                     (task-environment-driver
                                                      task-environment))
                              end-event-value)))))))))))
```

`Make-reactive-task` calls `proc` to yield a behavior `behavior` and a terminating event `end-event`. The behavior defined by the task returned is identical to `behavior` up until the first occurrence of `end-event` at which point the task will call the continuation with a newly created task environment.

13.7.3 Representing Preset Behaviors

Lula's sampling subsystem, and with it the rendering of reactive actions defined by Lulal program, is based on the idea of the *preset behavior*: a collection of parameter

settings. Lula's cue subsystem (see Section 11.14) maintains a preset for each cue, and Lulal extends this to animated presets.

The representation of preset behaviors is another aspect of the implementation of Lulal deserving some consideration. Ordinarily it would seem the ordinary behavior representation from the reactivity library described in the previous chapter would be just the right.

However, Lula uses a specialized representation for preset behaviors, primarily because they are at the juncture between the functional reactive subsystem and the imperative sampling engine. For instance, preset behaviors created from cues need to track all changes the user makes in the cue editor to assure instant feedback on changes. However, these changes happen imperatively and it is therefore difficult to forge the change history of a cue into the stream-based representation described in Section 12.4. Moreover, the implementation details for fades which must respect dimmer and fixture curves, among others, are easier to address in the sampling subsystem at the imperative level. Thus, it is more appropriate to embed the necessary domain-specific structural knowledge into the representation for preset behaviors. This effectively duplicates some implementation effort, but not enough to warrant complex interface machinery between the two. These considerations are similar to those in Fran [Elliott, 1998a, Elliott, 1999].

For the representation of the structural information about a preset behavior, Lula defines a set of record types which later become part of an instance of the `gbehavior<%>` interface described in the previous chapter in Section 12.5.6. Here is a description of the most important of them.

The most basic representation is for constant preset behavior:

```
(define-record-type :const-presetb
  (make-const-presetb preset)
  const-presetb?
  (preset const-presetb-preset))
```

The `:cue-presetb` record type represents cue-associated preset behaviors. The behavior follows all changes the user makes to the cue:

```
(define-record-type :cue-presetb
  (make-cue-presetb cue)
  cue-presetb?
  (cue cue-presetb-cue))
```

The `gbehavior<%>` interface requires `until` and `time-transform` methods which simply translate to the creation of instances of the `:until-presetb` and `:time-transform-presetb` record types.

```
(define-record-type :until-presetb
  (make-until-presetb behavior event)
  until-presetb?
  (behavior until-presetb-behavior)
  (event until-presetb-event))

(define-record-type :time-transform-presetb
  (make-time-transform-presetb behavior time-behavior)
  time-transform-presetb?
  (behavior time-transform-presetb-behavior)
  (time-behavior time-transform-presetb-time-behavior))
```

The `:restrict-presetb` record type is the implementation of the `restrict-with` primitive in Lulal:

```
(define-record-type :restrict-presetb
  (make-restrict-presetb behavior-1 behavior-2)
  restrict-presetb?
  (behavior-1 restrict-presetb-behavior-1)
  (behavior-2 restrict-presetb-behavior-2))
```

In the same vein, `:transformed-presetb` is responsible for implementing the `with-...` constructs in Lulal. Its behaviors specify a base behavior and a behavior of transformations to be applied to it.

```
(define-record-type :transformed-presetb
  (make-transformed-presetb behavior trafos-behavior)
  transformed-presetb?
  (behavior transformed-presetb-behavior)
  (trafos-behavior transformed-presetb-trafos-behavior))
```

Finally, the `:x-fade-presetb` type is for representing cross-fades. It takes preset behaviors `start-behavior` for the starting preset and `end-behavior` for the target preset, as well as the duration of the fade and a behavior representing the time left:

```
(define-record-type :x-fade-presetb
  (make-x-fade-presetb start-behavior duration time-left-behavior
                       end-behavior)
  x-fade-presetb?
  (start-behavior x-fade-presetb-start-behavior)
  (duration x-fade-presetb-duration)
  (time-left-behavior x-fade-presetb-time-left-behavior)
  (end-behavior x-fade-presetb-end-behavior))
```

There are corresponding representations for in/out fades and in/out fades with delay.

Based on these record types, Lula defines a `preset-behavior%` class which is an instance of `gbehavior<%>`. Each instance of `preset-behavior%` contains an instance of a representation record types; the class merely serves as an interface between the internal representation and the reactivity library.

13.7.4 Sampling Preset Behaviors

Finally, Lula needs to *sample* the preset behaviors created by the various preset sources in the system: a central driving loop keeps track of the various active preset behaviors, extracts a preset from each of them, combines these presets and feeds the resulting parameter settings to the fixtures view and the hardware interfaces.

Lula keeps a centralized time stream whose head is the current wallclock time. Each preset behavior has a loop which keeps sampling the behavior and communicating the resulting preset upwards, effectively moving continuous behaviors into the realm of the discrete. This upwards communication is not as trivial as it may seem at first, since a preset behavior might consist of other preset behaviors—it might, for example, be defined as the restriction of two other behaviors. Consequently, imperative code which simply sets slots in some array of parameter settings will not do, since the sampling settings must combine those same parameter settings with others later on.

Essentially, the easiest way to write the corresponding code would be along the following lines, showing only the case for constant preset behaviors:

```
(let ((representation (preset-behavior-representation preset-behavior)))
  (cond
```

```
((const-presetb? representation)
  (let ((preset (const-presetb-preset representation)))
   (let loop ()
     ⟨communicate preset upwards⟩
     (loop))))
...))
```

The ideal implementation paradigm for this is a generator-like mechanism where an expression can have a sequence of return values [Griswold and Griswold, 1996]. Lula contains an abstraction called *routines* for this purpose: A routine is a sort of subordinate coroutine. The running program creates a routine from a thunk. The program can yield control to the routine by calling the `resume` procedure on the routine. The routine then runs until it encounters a call to the `suspend` procedure. The routine then yields back control to the program, communicating its argument upwards. Lula contains two alternative implementations for routines, one based on continuations, the other based on threads.

With routines, the sampling code for constant behaviors looks as follows:

```
(cond
 ((const-presetb? representation)
   (let ((preset (const-presetb-preset representation)))
     (make-routine
      (lambda ()
        (let loop ()
          (suspend preset)
          (loop))))))
```

This avoids the need to keep explicit state between iterations of the sampling loop. The code for the other types of preset behaviors is analogous; some selected examples illustrate how it works.

```
((cue-presetb? representation)
  (let ((cue (cue-presetb-cue representation)))
    (make-routine
     (lambda ()
       (let loop ()
         (suspend (cue-preset cue))
         (loop))))))
```

The sampling of some preset behaviors first requires sampling some other component behaviors and assembling a preset from their results. Transformed are an example:

```
((transformed-presetb? representation)
  (let ((behavior (transformed-presetb-behavior representation))
        (trafos-behavior
          (transformed-presetb-trafos-behavior representation)))
    (let ((behavior-routine (preset-behavior->routine behavior)))
      (make-routine
       (lambda ()
         (let loop ((trafos-stream (at trafos-behavior (the-time-stream))))
           (let ((trafos (stream-car trafos-stream)))
             (suspend (preset-apply (resume behavior-routine) trafos))
             (loop (stream-cdr trafos-stream)))))))))
```

This routine first creates a routine for sampling `behavior`. Moreover, it uses the reactivity library's `at` procedure to obtain a sample stream for `trafos-behavior`.

(`The-time-stream` returns a stream of sampling times starting at wallclock time.) At each iteration of the loop, the routine resumes `behavior-routine`, transforms the result and suspends it back to the program.

Sometimes, it is necessary to switch sampling from one behavior to another, most notably with `until`-constructed reactive behaviors. Here is their implementation. First, routine creates another routine for sampling `behavior`:

```
((until-presetb? representation)
  (let ((behavior (until-presetb-behavior representation))
        (event (until-presetb-event representation)))
    (let ((behavior-routine (preset-behavior->routine behavior)))
      (make-routine
       (lambda ()
```

The sampling loop must watch out for occurrences of `event`. It obtains a stream of possible occurrences from the `occurrences` procedure from the reactivity library. When that stream is at its end, the behavior is identical to `behavior` until the end of time. In that case, the routine returns another routine to take its place instead of a preset:

```
          (let loop ((event-occs-stream
                      (occurrences event (the-time-stream))))
            (cond
             ((stream-null? event-occs-stream)
              (suspend behavior-routine))
```

When the routine finds an event occurrence, it samples `behavior` one last time and then switches over to a newly created routine for sampling the new behavior:

```
             ((stream-car event-occs-stream)
              => (lambda (occurrence)
                   (suspend (resume behavior-routine))
                   (let* ((behavior (tforce (occurrence-tpromise occurrence)))
                          (behavior-routine (preset-behavior->routine behavior)))
                     (suspend behavior-routine))))
```

Finally, if the event has not occurred yet, the routine just keeps on sampling `behavior`:

```
            (else
              (suspend (resume behavior-routine))
              (loop (stream-cdr event-occs-stream)))))))))))
```

With the actual sampling code in place, Lula's sampling subsystem is based on the concept of *drivers*: a driver is associated with a source of parameter settings; each driver defines a routine that keeps resuming presets or new routines. At its core, the sampling subsystem calls the `kick-driver!` routine which obtains a sample from a driver:

```
(define (kick-driver! driver)
  (let loop ()
    (let ((routine-or-preset (resume (driver-routine driver))))
      (cond
       ((routine? routine-or-preset)
        (set-driver-routine! driver routine-or-preset)
        (loop))
       ((preset? routine-or-preset)
        (set-driver-last-preset! driver routine-or-preset)
        (inject-preset! routine-or-preset))))))
```

`Kick-driver!` keeps resuming the routine until it produces a preset. At that point, it stashes the resulting preset for use by subsequent tasks, and calls `inject-preset!` to supply both the fixture views and the hardware interfaces with the preset data.

Part VII

Closing Arguments

Oiga, amigo. If ever somethin'
don't feel right to you, remember what
Pancho said to The Cisco Kid ...
'Let's went, before we are dancing at
the end of a rope, without music.'
—SAILOR in *Wild at Heart*

I did not start to write Lula to make a point originally. Its inception was as much accident as it was intention, and it grew to the point where it is today mainly because of the requirements which grew out of daily use, as well as ignorance of much of the work done by the designers of commercial consoles.

In this part, I review the work done so far, and summarize the contributions of this dissertation. The final chapter is an outlook on future work.

Chapter 14

Assessment

I also want to
thank you fellas, you've taught me
a valuable lesson in life.
LULA!!!!
—Sailor in *Wild at Heart*

This dissertation has described Lula, a system for computer-assisted lighting design and control. Lula's advantage over conventional lighting consoles rests on two cornerstones: its user interface and its implementation. Without the former, Lula is just one more lighting console. Without the latter, it would not have been possible to implement Lula in the time available. This chapter looks back on the Lula project and tries to assess the validity of the techniques used during its lifetime so far.

14.1 Lula in the Real World

Assessing the effectiveness of Lula's user interface is inherently difficult: An objective investigation would compare Lula side-by-side with another high-end lighting console as applied to a single project under realistic conditions to achieve comparable results. This is impossible in practice for several reasons:

- "Realistic conditions" for a lighting console means a stage with lots of fixtures—in the case of Lula, with at least some multi-parameter fixtures. Such stages are not generally available for experimentation—they are usually in use.

- A "typical user" for a system like Lula hardly exists. Lighting designers differ in the methodologies they employ; lighting operators often define their expertise by the selection of consoles they know.

- If the same person should redo a project to evaluate another console, the environmental conditions will have changed to the point which makes the result of the comparison meaningless: the designer and operator know much more about the final result after the first design. The comparison would only have some meaning if the second run would take *longer*.

- Arguably, the objective is not really to get the job done faster, but rather to get it done better. However, "good" and "better" are elusive qualities in what is effectively an art form.

- Obtaining sufficient data for anything approaching an empirical quantitative analysis is completely out of the question. Lighting design for single show typically takes hours, often days.

Having stated some of the necessary limitations of the assessment, here are some data points from the real world:

- The fundamental concept underlying Lula—componential lighting designs—grew directly out of the difficulties I had had with conventional consoles in a number of venues. Many hair-raising situations I have lived through—never-ending sessions of calling out channel numbers and percentages, pervasive last-minute changes—would objectively not have happened if I had had Lula available to me.

- I myself have used Lula extensively, both in the Brecht-Bau theater and on tour. I have used it on a variety of small and medium-sized stages. Besides the Brecht-Bau theater, these include the Foyer U3 in Reutlingen, the Metropol in Munich, and the Theaterhaus in Stuttgart.

  In my own experience, designing lighting with Lula is *fast*. In the Brecht-Bau Theater, Lula has cut the entire time I have needed for lighting shows in half—including hanging and focusing. I have had the opportunity to re-light the same show on different stages, with and without Lula. Again, the time spent operating the console is easily cut in half, no matter if it is me or someone familiar with the local stage and lighting console is operating the lights.

- With a touring production, when the programming from a previous venue is still available, the time savings are considerably greater. I had the opportunity to take Lula with me on U34's production of Peter Shaffer's *Equus* in 1999. I cut down time spent on lighting by about a factor of four in those venues where I was able to use Lula. More importantly, some of the shows would not have gotten off the ground in time for the curtain with another console.

- Audience members who saw *Equus* in several venues reported that the lights were much better in those where we used Lula.

- I give occasional workshops on the use of Lula to the amateur lighting designers working at the Brecht-Bau theater. The workshop—usually a group of about 6—takes at most three hours. After that time, the users are usually completely on their own. Questions are rare and mostly concern operative details rather than conceptual issues.

The accounts reflect mainly the experience with Lula 3 which was limited to theatrical lighting. The effect subsystem and Lulal have not received any exposure to practitioners except myself. Still, Lulal and its library functionality subsume the effect subsystems of all conventional consoles whose documentation was available to me. The ease of operation is at least comparable to these consoles, arguably much better.

Lula breaks with the tradition of conventional console design, which still treats a lighting console essentially like an automated version of a fader board. This historical baggage accounts for the complexity of the user interface of most of today's popular high-end consoles. Their manuals often comprise several hundred pages and still omit essential aspects of the consoles' functionality.

In contrast, Lula's fundamental concepts are few in number which reduces the conceptual overhead of its operation. Moreover, these concepts relate directly to the structure of the design process, or at least a common form of it. I conjecture that these aspects of the design of Lula form its principal advantage. The application of

modern user-interface design techniques, also a part of Lula's development, mainly reflect the structure of the system rather than impose a structure of their own. For this reason as well as reasons of space, this dissertation does not contain a discussion of user-interface issues.

14.2 Modelling

The essential ingredients for Lula's model of lighting builds on three essential ingredients:

Algebraic modelling Algebraic methods were instrumental in designing Lula's cue model. Whereas I designed the simple HTP-only model in Lula 1—3 purely ad-hoc, I actually derived the specification in Chapter 11 by the algebraic methods set forth there, in roughly that order. The domain-theoretic interpretation formed the basic intuition for the design of the support for multi-parameter fixtures. Consequently, Lula's cue model would not have been near as powerful as it is without of these methods.

The lack of a proper design foundation in conventional consoles emphasize the validity of this method. Specifically, the two hierarchical consoles on the market have no explicit machinery in place to deal with semantic issues such as the composition of transformations or conflict avoidance.

Domain-specific languages The specification for cues is effectively a small domain-specific language to create static lighting components. The graphical cue editor obscures this aspect of the cue subsystem, but it would be perfectly feasible to offer a more general programming interface to cues.

Furthermore, whereas the effect subsystems in conventional consoles are purely ad-hoc, Lula builds on a strong and powerful semantic foundation: functional reactive programming—effectively an embedded domain-specific language—provides a common substrate for all light change and animation in Lula.

Moreover, Lula puts the expressive power of functional reactive programming in the hands of the user through Lulal, a standalone domain-specific language. While Lulal is powerful, the novice user does not have to deal with it to create animated light—the system provides an extensible library of common task for direct use.

Stratified design Finally, the core of Lula constitutes a classic stratified design: fixtures form the lowest level, cues build upon them. Presets build upon cues. Preset behaviors build upon presets on cues. The sampling subsystem finally builds upon preset behaviors. The insulation between these layers if reflected in Lula's module structure, and it would be reasonably easy to replace any layer independently from the others. In fact, this happened a number of times during the development process.

14.3 Quantifying the Development Process

Assessing the development process of the Lula software is subject to similar limitations as assessing the effectiveness of its user interface: there is just no basis to comparison which would allow me to say that different techniques would not have produced similar results. In the light of these problems, it is at least instructive to look at some numbers.

Table 14.1 shows the development time needed for each successive version of Lula. The numbers are necessarily approximate. Except for the first version, I

Lula 1	1
Lula 2	1
Lula 3	2
Lula 4 (current)	8
Lula 4 (estimated final)	12

Table 14.1: Development times (in Mike-weeks).

never had even a contiguous week of full time available to spend on Lula. The most important number is the first: it took me almost exactly a week to finish the first version. I had not worked with DrScheme or known about its GUI toolkit before. Still, Lula 1 was fully functional and fairly stable—it ran nonstop at CeBIT 1997 for six nine-hour days without a single crash, and was also used to light a production of *Who's afraid of Virgina Woolf?* in the Brecht-Bau theater. There was one glitch on the third night: the hardware overheated. Still, I think it is safe to say that only the use of advanced development techniques could enable anyone to write the software this fast.

	Core	Library	Total
Lula 1	6000	0	6000
Lula 2	5400	1000	6400
Lula 3	11000	3000	14000
Lula 4 (current)	17000	7000	24000
Lula 4 (estimated final)	20000	10000	30000

Table 14.2: Code size of successive Lula versions (in loc).

Table 14.2 shows the (absolute) line counts of the Lula code in successive versions, separating between code specific to lighting and library code which could be useful to other applications. Between Lula 1 and Lula 2 the code consolidated which accounts for the only slight increase in size. Lula 3 and Lula 4 presented significant functional enhancements. Particularly, the support of multi-parameter fixtures and animated lighting in Lula 4 is taking its toll. Studies seem to suggest that Scheme code is about 1/3 to 1/10 of the size of C++ code with comparable functionality [Hudak and Jones, 1994]. A factor of 1/2 to 1/5 compared with Java seems realistic. I conjecture the sheer typing work would have failed to meet the deadline of Lula 1, for instance.

Another quantitative aspect of Lula is its reliability. This is even harder to assess objectively. The bugs found by end users were precious few, less than six at the time of writing of this dissertation. The only software bug which ever occurred during a show was not in Lula itself, but rather in the substrate implementing DrScheme's editor functionality which is written in C++.

14.4 Reviewing Tools

In retrospect, the choice of development tools—Scheme and DrScheme—has been exceedingly appropriate for the task at hand. It is worthwhile to reconsider the most important techniques cites in Chapter 8:

Automatic memory management It is sad that memory management is still the subject of heated discussion; its benefits have been known for decades, and

the implementation techniques have progressed to the point where garbage collectors can compete with the best hand-tailored allocation and memory management strategies.

For an application with the size of Lula which, due to its highly interactive nature *has* to perform dynamic memory management, garbage collection is indispensable. It is simply not practical to implement manual storage management to the point that no space leaks remain which is necessary to ensure the reliable operation of the system.

Higher-order programming Higher-order programming is pervasive in Lula, so it is difficult to say what the code would look like without it. Many implementation choices—the representation of reactive behaviors, for instance—are made possible only by the availability of higher-order procedures in Scheme and their notational succinctness.

Approaches such as this one lose much of their appeal if higher-order procedures are only available through some surrogate mechanism such as anonymous classes in an object-oriented language. This is actually an often-overlooked aspect of programming language design: Lisp and Scheme programmers often claim that "syntax doesn't matter" because their languages basically allow them to have any syntax they want within the confines of parenthesized sexprs. However, most other programming languages do not have extensible syntax, and consequently the programmer is stuck with the syntax at hand. Given this, even though a particular programming paradigm may be expressible in the language, the notational inconvenience may make the paradigm considerably less attractive. This chasm is the cause for many misunderstandings between programming language evangelists.

Functional programming Wherever feasible, I have used functional data structures in Lula. This was not always the case: versions of Lula before Lula 4 used a fair number of imperative data structures: presets were array-based, as well as all the representation of all patching information. Executing light fades were represented as arrays of state automata. The list goes on.

Lula 4 contains not a single array, and updates to record types have been confined to a few well-chosen places. This has led to the lifting of a number of implementation limits—the number of fixtures or number of channels supported by the application, for instance. Moreover, I was able to remove a significant amount of thread synchronization code, and thus reduce the opportunities for race conditions and other bugs inherently related to the use of imperative data structures.

Data-directed programming In data-directed programming or its incarnation, message-passing style, many software practitioners will recognize a familiar part of the dominant variant of object-oriented programming also known as *dynamic dispatch*. In the words of Jonathan Rees:

> Object-oriented programming isn't a single idea but a hodgepodge of several
>
> - references to changing state (object = variable = pointer)
> - interface inheritance implementation inheritance (code re-use)
> - dynamic dispatch
>
> The "object" concept is sophistry and should be deconstructed whenever it's used.

Lula code uses DrScheme's object system extensively. However, most of these uses make no use of inheritance; the merely use objects as a convenient notation for data-directed programming.

Parametric inheritance Inheritance only shows up in relation to Lula's graphical user interface. This is unavoidable as DrScheme's GUI toolkit is based on objects and inheritance.

However, all inherent uses of inheritance within Lula are parametric (see Section 8.7), and are confined to the construction of the graphical user interface. Mixins have been very useful in Lula's development. They implement a number of useful features for Lula's GUI components, among them the resurrection of window geometry, the correct setting of frame icons, and the creation and appropriate arrangement of menu items.

Concurrent programming Finally, Lula's model of execution is inherently concurrent: the program needs to drive the interface hardware while still being responsive to user actions. Multiple sources can provide parameter settings simultaneously. Several actions can execute concurrently. Implementing the system without concurrency support in the programming language would have been much harder and certainly impossible in the time span available.

14.5 Objections

Lighting design and control is a field full of opinionated practitioners. So, naturally, there are a number of criticisms of Lula both from the field and from the outside.

> **Objection 1:** "Experienced lighting designers and operators are used to the conventional lighting boards. They will not be able to adjust to Lula's way of doing things."

This is certainly valid criticism, but also sort of a self-fulfilling prophecy: If users would stick with systems they know, progress would never happen. I have invested considerable time in providing at least some terminology and some controls within the software familiar to users of conventional consoles. When I talk with professional lighting designers, they usually have little trouble understanding the concepts—after all, they are based on actual design process. The step from there to actual operation of the Lula is small. All it takes is the will to make that step.

> **Objection 2:** "Most of time spent during lighting design is spent hanging and focusing the fixtures, putting in gels, and so on, not with programming the console."

This is probably true in today's theatrical environments. But still, considerable time is spent at the console. Every hour saved counts. Moreover, I predict that this objection will become less valid in the future with the advent of multi-parameter lights. More and more of the work now done manually at the fixtures will be remote-controlled from the console in the future.

> **Objection 3:** "An effective lighting console requires a special control board. A PC keyboard and a mouse or trackball slows down the operator."

This issue is closely related with the design of the board: Many consoles were inherently designed to require a bulky control board. Lula was not, and consequently it

does well without one. However, in some situations, especially with highly dynamic lighting, Lula could benefit from more easily identifiable buttons. There is nothing in the software architecture which prevents implementing support for an external add-on control board.

Chapter 15

Future Work

Johnnie ... That's the past ... We gotta get on to our future, sugar!
—Marietta in *Wild at Heart*

Like any project, Lula is far from finished. This holds at the time of writing of this dissertation, and will probably hold true forever. This chapter outlines some possible directions for future work.

15.1 Lula 4

Lula 4 needs some polishing before it is ready for release. Most of the remaining tasks are small: numerous cosmetic improvements to the user interface, driver support for more hardware, and bug fixes.

Lula also needs a comprehensive library of fixture types ready for patching. A domain-specific language for specifying them would be nice. The set of available transformation types is far from complete. Also, the effect subsystem and the Lulal language require better integration with the rest of the system. The system needs a larger base library of common effects. A freehand shape editor would also be nice.

Lula's user interface needs one more fundamental addition: its support for cues is mainly constructive. However, with a finished cue, the designer might point at a fixture at say "Why is this fixture on?" The answer may be hidden in the cue hierarchy, and Lula presently does not help much in finding it. Presenting the necessary information in a useful way requires defining a formal notion of *responsibility* for the cue specification and implementing a user interface for it. The former has been done, the latter has not.

15.2 Lula 5

Lula 4 will hardly be the end-all of lighting consoles. While Lula 4 focuses on conceptual modelling of the lighting design, and the use of abstraction, the plan for Lula 5 is to assist the operator in dealing with the concrete: While the selection and assignment of fixtures is a straightforward matter in Lula 4, fixtures are still essentially numbers to Lula.

Therefore, Lula 5 will provide modelling for the stage geometry, so that the user can select fixtures by pointing at a symbol on the groundplan. The next step is the extension of the groundplan into the three-dimensional, and full geometric modelling of multi-parameter fixtures, including some sort of three-dimensional rendering of the resulting looks. This is primarily useful in show lighting on stages with fixed

riggings. In theater environments, simulation the look of cues is of limited usefulness because of the importance of (possibly variable) set pieces, textures, and nuance is general.

15.3 Add-On Hardware

Lula's usability could probably benefit from a specialized control console, especially for continuous controls. The difficulty with specialized controls is always determining the mapping to input parameters for the software. A fixed mapping is easiest to remember to the user but limits flexibility. Many commercial consoles use touchscreens. This approach would probably also work well for Lula.

One area that merits special attention is the advent of wearable computers and wireless networking. Many commercial consoles already come with limited remote controls so that the operator does not need to constantly run back and forth between the stage and the console. With a wearable computer, it becomes possible to offer the complete functionality of the software. It also remains to be investigated how speech input can be useful in lighting.

15.4 General Show Control

Running a show involves other aspects besides lighting. Specifically, the problems of sound playback are closely related to those of lighting playback. Light and sound effects often require coordination. To this end, many commercial lighting consoles come with MIDI inputs to take "Go" signals from some music source. The resulting setups are often awkward, and still mostly require the operator to press buttons on two consoles. Hence, support for playing back sounds—CD tracks or digitized sound effects—is desirable. This has some nice side-effects; for instance, Lula could check that the operator has really inserted the right CD for an upcoming sound playback.

A prototype extension of Lula 2 already includes support for playing back CD tracks. Lula's present present action architecture easily supports this feature. The extension just needs backporting and some polishing up. It remains to be seen whether functional reactive programming could also help design sound effects, say for designing sound envelopes.

15.5 Lessons for Tool Support

The previous chapter has already identified how a programming language can support effective application development by providing a number of programming paradigms. It is not always sufficient for a programming language implementation to satisfy the general language-level requirements of these paradigms. In some areas, performance issues are sufficiently important to have a major impact on the usefulness of certain paradigms. Some of them warrant work in the context of Lula development:

Garbage collection Garbage collection has long been stigmatized as too inefficient for practical use. Even recently published computer science textbooks insist that garbage collection inherently hurts performance. The truth is that most implementations of garbage collection are poor, especially in freely available software. (Emacs being a notorious example.) DrScheme's garbage collector is currently improving significantly, but still causes noticeable pauses. An incremental collector would help satisfy Lula's modest real-time requirements better.

Lightweight continuations One of the distinguishing characteristics is its support for *first-class continuations*: a running program can reify the current continuation at any time via the `call-with-current-continuation` primitive. The reified continuation has unlimited extent; the program can invoke it multiple times. Continuations are useful for implementing a number of programming paradigms, among them exception systems, various other non-local control structures [Dybvig, 1996], thread systems, and monads [Filinski, 1994].

However, to be truly practical for these applications, continuations need to efficient in space in times. This requirement basically excludes traditional stack implementations of continuations. More efficient implementations have been known since the late 80s [Clinger *et al.*, 1999], but have not made their way into many implementations yet. Unfortunately, DrScheme is not presently among them.

Lightweight continuations would have, for example, simplified the implementation of routines (see Section 13.7.4).

Lightweight threads The term "threads" by now comprises a wide range of process-like abstractions. The implementation of threads is closely related to the representation of a continuation, as a thread is usually characterized as a process which shares state with other threads but has its own continuation. Implementations of threads differ greatly in the cost of their representation, ranging from about 100 bytes per thread in SML/NJ to about 2 Megabytes for Linux operating threads.

Truly lightweight threads allow, for instance, attaching a thread to every single component of the graphical user interface, thereby greatly simplifying the encapsulation of state [Gansner and Reppy, 1993]. DrScheme threads take up about 20 Kilobytes each which makes them impractical for this purpose. Consequently, I had to undertake significant efforts to reduce the number of threads in Lula. With a more lightweight thread system, it would also be feasible to implement a much richer library of synchronization primitives such as CML [Reppy, 1988, Reppy, 1999].

15.6 Art

At the end of the day, what counts with a lighting system is the show the audience sees. Thus, ultimately, the goal of the Lula Project is to improve the quality of lighting design by improving the quality of the tools available to the designers—to further art. Of course, a tool which saves time is already useful to this end: the designer use the time gained to perfect the design. Still, I think that especially animated lighting has not reached its potential, primarily because of the limited functionality of the available control systems. I predict that, as better tools such as Lula emerge, a true discipline of choreographed light will emerge. I look forward to that time.

Bibliography

[Abelson *et al.*, 1996] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., second edition, 1996.

[ASMR2D2, 2000] ASM-Steuerungstechnik GmbH, Wünnenberg-Haaren, Germany. *R2-D2 exclusive 1024 DMX512 Lighting Controller Benutzerhandbuch*, build 2541 edition, 2000.

[Barendregt, 1990] Henk P. Barendregt. Functional programming and lambda calculus. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science—Formal Models and Semantics*, volume B, chapter 7. Elsevier Science Publishers, 1990.

[Bentley, 1986] Jon Louis Bentley. Programming pearls—little languages. *Communications of the ACM*, 29(8):711–721, August 1986. Description of the *pic* language.

[Boussinot and Susini, 1998] Frédéric Boussinot and Jean-Ferdy Susini. The SugarCubes Tool Box: A reactive Java framework. *Software—Practice & Experience*, 28(14):1531–1550, December 1998.

[Boussinot, 1991] Frédéric Boussinot. Reactive C: An extension of C to program reactive systems. *Software—Practice & Experience*, 21(4):401–428, April 1991.

[Cejtin *et al.*, 1995] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995.

[Chambers *et al.*, 2000] Craig Chambers, Bill Harrison, and John Vlissides. A debate on language and tool support for design patterns. In Tom Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 277–289, Boston, MA, USA, January 2000. ACM Press.

[ClayPakyGoldenScan, 2000] ClayPaky, Pedrengo, Italy. *Golden Scan "HPE"*, 2000. Available electronically as `http://www.claypaky.it/acrobat/Golden\%20S\%20HPE\%20ING.zip`.

[ClayPakyTigerCC, 2000] ClayPaky, Pedrengo, Italy. *Tiger C. C.*, 2000. Available electronically as `http://www.claypaky.it/acrobat/Tiger\%20CC\%20Ing.zip`.

[Clinger *et al.*, 1999] William D. Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 1(12):7–45, April 1999.

[Damas and Milner, 1982] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[DINDMX, 1997] *Bühnenlichtstellsysteme, Teil 2: Steuersignale*. Beuth Verlag, Berlin, 1997. DIN 56930-2.

[DMX512-1990, 1990] *DMX512/1990, Digital Data Transmission Standard for Dimmers and Controllers*. United States Institute for Theatre Technology, 1990.

[DMX512-2000, 2000] Entertainment Services and Technology Association, United States Institute for Theatre Technology, Inc. *BSR E1.11, Entertainment Technology — USITT DMX512-A Asynchronous Serial Data Transmission Standard for Controlling Lighting Equipment and Accessories*, 1.6 edition, 2000. DRAFT.

[Dybvig, 1996] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 2nd edition, 1996.

[Elliott and Hudak, 1997] Conal Elliott and Paul Hudak. Functional reactive animation. In Mads Tofte, editor, *Proc. International Conference on Functional Programming 1997*, Amsterdam, The Netherlands, June 1997. ACM Press, New York.

[Elliott, 1997] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997. USENIX.

[Elliott, 1998a] Conal Elliott. From functional animation to sprite-based display (expanded version). Technical Report MSR-TR-98-28, Microsoft Research, Microsoft Corporation, Redmont, WA, October 1998. Available electronically as `http://www.research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=19\%0`.

[Elliott, 1998b] Conal Elliott. Functional implementations of continuous modeled animation. In Catuscia Palamidessi and Hugh Glaser, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '98)*, number 1490 in Lecture Notes in Computer Science, Pisa, Italy, September 1998. Springer-Verlag.

[Elliott, 1998c] Conal Elliott. Functional implementations of continuous modeled animation (expanded version). Technical Report MSR-TR-98-25, Microsoft Research, Microsoft Corporation, Redmont, WA, July 1998. Available electronically as `http://www.research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=16\%4`.

[Elliott, 1998d] Conal Elliott. An imperative implementation of functional reactive animation. Available electronically as `http://www.research.microsoft.com/~conal/newFran/design/`, December 1998.

[Elliott, 1999] Conal Elliott. From functional animation to sprite-based display. In Gupta [1999].

[ESTA2000, 2000] ACN developer tutorial. `http://www.esta.org/tsp/PLASA_2000_Tutorial_4up.pdf`, September 2000. PLASA Show.

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

[Felleisen *et al.*, 1998] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The DrScheme project: An overview. *SIGPLAN Notices*, 33(6):17–23, June 1998.

[Field and Harrison, 1988] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[Filinski, 1994] Andrzej Filinski. Representing monads. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, OG, January 1994. ACM Press.

[Findler and Flatt, 1998] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In Hudak [1998].

[Finne and Peyton Jones, 1995] Sigbjørn Finne and Simon Peyton Jones. Composing Haggis. In *Proc. 5th Eurographics Workshop on Programming Paradigms in Graphics*, Maastricht, NL, September 1995.

[Fitt and Thornley, 1992] Brian Fitt and Joe Thornley. *Lighting by Design*. Focal Press, Oxford, England, 1992.

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

[Flatt and Findler, 2000a] Matthew Flatt and Robert Bruce Findler. *PLT Framework: GUI Application Framework*. Rice University, University of Utah, August 2000. Version 103.

[Flatt and Findler, 2000b] Matthew Flatt and Robert Bruce Findler. *PLT MrEd: Graphical Toolbox Manual*. Rice University, University of Utah, August 2000. Version 103.

[Flatt *et al.*, 1998] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In Luca Cardelli, editor, *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, pages 171–183, San Diego, CA, USA, January 1998. ACM Press.

[Flatt *et al.*, 1999] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems. In Peter Lee, editor, *Proc. International Conference on Functional Programming 1999*, pages 138–147, Paris, France, September 1999. ACM Press, New York.

[Flatt, 2000] Matthew Flatt. *PLT MzScheme: Language Manual*. Rice University, University of Utah, August 2000. Version 103.

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

[Foley *et al.*, 1996] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C*. Addison-Wesley, second edition, 1996.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gansner and Reppy, 1993] Emden R. Gansner and John H. Reppy. A multi-threaded higher-order user interface toolkit. In L. Bass and P. Dewan, editors, *User Interface Software*, volume 1 of *Trends in Software*, chapter 4, pages 61–80. John Wiley & Sons Ltd, 1993.

[Gerthsen *et al.*, 1989] Christian Gerthsen, Hans O. Kneser, and Helmut Vogel. *Physik*. Springer-Verlag, Berlin, Heidelberg, New York, 1989.

[Gillette, 1989] J. Michael Gillette. *Designing with Light*. Mayfield Publishing Company, Mountain View, CA, second edition, 1989.

[Gogolla *et al.*, 1984] Martin Gogolla, Klaus Drosten, Udo Lipeck, and Hans-Dieter Ehrich. Algebraic and operational semantics of specifications allowing exceptions and errors. *Theoretical Computer Science*, 31(3):289–313, December 1984.

[Grab, 1995] Till Grab. Anforderungsprofil für Lichtregieanlangen in kleineren und mittleren Theatern und Veranstaltungsorten. Master's thesis, Technische Fachhochschule Berlin, 1995.

[Griswold and Griswold, 1996] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, 3rd edition, 1996.

[Gunter and Scott, 1990] Carl A. Gunter and Dana S. Scott. *Semantic Domains*, volume B of *Handbook of Theoretical Computer Science*, chapter 12. Elsevier Science Publishers, Amsterdam, 1990.

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

[Gupta, 1999] Gopal Gupta, editor. *Practical Aspects of Declarative Languages, Proceedings of the First International Workshop, PADL '99*, number 1551 in Lecture Notes in Computer Science, San Antonio, Texas, USA, January 1999.

[Hailperin *et al.*, 1999] Max Hailperin, Barbara Kaiser, and Karl Knight. *Concrete Abstractions*. Brooks/Cole, 1999.

[Haskell98, 1998] Haskell 98, a non-strict, purely functional language. `http://www.haskell.org/definition`, December 1998.

[HighEndColorPro, 1999] High End Systems, Inc., Austin, Texas. *Color Pro User Manual*, version 1.1 edition, September 1999. Available electronically as `ftp://ftp.highend.com/pub/Products/ColorPro/colorpro.pdf`.

[HighEndCyberlight, 1996] High End Systems, Inc., Austin, Texas. *Cyberlight User Manual*, version 2.0 edition, May 1996. Available electronically as `ftp://ftp.highend.com/pub/Products/Cyberlight/cyber200.exe`.

[HighEndDataFlashAF1000, 1997] High End Systems, Inc., Austin, Texas. *Dataflash AF1000 Xenon Strobe User's Manual*, December 1997. Available electronically as `ftp://ftp.highend.com/pub/Products/AF1000/af1000.pdf`.

[HighEndStatusCue, 1997] High End Systems, Inc., Austin, Texas. *Status Cue User's Manual*, May 1997. Available electronically as `ftp://ftp.highend.com/pub/Products/StatusCue/Manual/statq.pdf`.

[HighEndStudioBeamPC, 2000] High End Systems, Inc., Austin, Texas. *High End Studio Beam PC User Manual*, version 1.0 edition, June 2000. Available electronically as `ftp://ftp.highend.com/pub/Products/StudioBeamPC/Manual/BeamPC.pdf`.

[Hindley, 1969] J. R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[Hudak and Jones, 1994] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ..., an experiment in software prototyping productivity. Technical report, Yale University, Department of Computer Science, New Haven, CT 06518, July 1994.

[Hudak *et al.*, 1996] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation — an algebra of music —. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[Hudak, 1998] Paul Hudak, editor. *International Conference on Functional Programming*, Baltimore, USA, September 1998. ACM Press, New York.

[Hudak, 2000] Paul Hudak. *The Haskell School of Expression: learning functional programming through multimedia*. Cambridge University Press, 2000.

[Hughes, 1990] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 2, pages 17–42. Addison-Wesley, 1990.

[Huntington, 2000] John Huntington. *Control Systems for Live Entertainment*. Focal Press, 2000.

[Kelsey and Rees, 1995] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[Kelsey *et al.*, 1998] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[Kelsey, 1999] Richard Kelsey. SRFI 9: Defining record types. `http://srfi.schemers.org/srfi-9/`, September 1999.

[Klaeren, 1983] Herbert Klaeren. *Algebraische Spezifikation — Eine Einführung*. Lehrbuch Informatik. Springer-Verlag, Berlin-Heidelberg-New York, 1983.

[Krasner and Pope, 1988] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, August/September 1988.

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

[LEE Filters, 2000] LEE Filters. Lighting filters. `www.leefilters.com`, 2000.

[MAGrandMA, 2000] MA Lighting Technology GmbH, Waldbüttelbrunn, Germany. *grandMA User's Manual*, version 2.0 edition, August 2000. Available electronically as `http://malighting.com/products/pdf/grandMA/GM_manu_english.zip`.

[Marks, 1998] Patrick Marks. *Avolites Diamond I and III—Operators Manual*. Avolites, England, 05-05-98 edition, July 1998. Available electronically from `http://www.avolites.btinternet.co.uk/software/download.htm`.

[MartinCase, 2000] Martin Professional AS, Århus, Denmark. *Martin Case Manual*, version 7.20 edition, April 2000. Available electronically from `http://www.martin.dk/service/`.

[MartinMac600, 2000] Martin Professional AS, Århus, Denmark. *MAC 600/E user manual*, 2000. Available electronically as `http://www.martin.dk/service/manuals/mac600.pdf`.

[MartinRoboScan918, 1999] Martin Professional AS, Århus, Denmark. *RoboScan Pro 918 user manual*, 1999. Available electronically as `http://www.martin.dk/service/manuals/roboscanpro918.pdf`.

[MAScanCommander, 1996] MA Lighting Technology GmbH, Waldbüttelbrunn, Germany. *Scancommander User's Manual*, version 4.x edition, October 1996. Available electronically as `http://malighting.com/products/pdf/scengl.pdf`.

[MIDI, 1996] *MIDI 1.0 detailed specification: Document version 96.1*. MIDI Manufacturers Association, 1996.

[Milner *et al.*, 1997] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Mitchell, 1999] Tim Mitchell. *Avolites Sapphire 2000 Operator's Manual*. Avolites, England, July 1999. Available electronically from `http://www.avolites.btinternet.co.uk/software/download.htm`.

[Moggi, 1988] Eugenio Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, 1988.

[Moody, 1998] James L. Moody. *Concert Lighting*. Focal Press, second edition, 1998.

[Okasaki, 1998] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Peterson and Hager, 1999] John Peterson and Greg Hager. Monadic robots. In *2nd Conference on Domain-Specific Languages*, Austin, Texas, USA, October 1999. USENIX. `http://usenix.org/events/dsl99/index.html`.

[Peterson *et al.*, 1999a] John Peterson, Greg Hager, and Paul Hudak. A language for declarative robotic programming. In Yuan F. Zheng, editor, *Proceedings of the International Conference on Robotics and Automation Information 1999*, Detroit, Michigan, May 1999. IEEE Press.

[Peterson *et al.*, 1999b] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In Gupta [1999].

[Peyton Jones *et al.*, 1999] Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: weak pointers and stable names in Haskell. In Pieter Koopman and Chris Clack, editors, *Implementation of Functional Languages*, Lochem, The Netherlands, September 1999. LNCS 1868.

[Pucella, 1998] Riccardo Pucella. Reactive programming in standard ml. In *IEEE International Conference on Computer Languages, ICCL 1998*, Chicago, USA, May 1998. IEEE Computer Society Press.

[Reid, 1987] Francis Reid. *The Stage Lighting Handbook*. A & C Black, London, England, third edition, 1987.

[Reid, 1993] Francis Reid. *Discovering Stage Lighting.* Focal Press, 1993.

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

[Reppy, 1999] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[Rosco Laboratories, 2000] Rosco Laboratories. filters. `www.rosco.com`, 2000.

[RoscoHorizon98,] Rosco Entertainment Technology, Portland, Oregon. *Horizon 98 User Manual*, build 680 edition. Available electronically as `ftp://rosco-et.com/pub/web5510c/hz-manual-680.pdf`.

[Sage, 2000] Meurig Sage. FranTk — a declarative GUI language for Haskell. In Philip Wadler, editor, *Proc. International Conference on Functional Programming 2000*, pages 106–117, Montreal, Canada, September 2000. ACM Press, New York.

[Sandström, 1997] Ulf Sandström. *Stage Lighting Controls.* Focal Press, 1997.

[Scholz, 1998] Enno Scholz. Imperative Streams: A monadic combinator library for synchronous programming. In Hudak [1998], pages 261–272.

[Shelley, 1999] Steven Louis Shelley. *A Practical Guide to Stage Lighting.* Focal Press, Oxford, England, 1999.

[Steele Jr. and Gabriel, 1993] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of lisp. In Jean E. Sammet, editor, *History of Programming Languages II*, pages 231–270. ACM, New York, April 1993. SIGPLAN Notices 3(28).

[Steele, Jr., 1998] Guy L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, October 1998.

[StrandGeniusPro, 2000] Strand Lighting, Middlesex, UK. *GeniusPro Lightpalette Operator's Guide*, v2.4 edition, March 2000. Available electronically as `http://www.strandlight.com/eu/files/Manuals/430-550/lp2_4/guide.pdf`.

[StrandTracker, 1998] Strand Lighting, Middlesex, UK. *Tracker Moving Light Software Operator's Manual*, v2.2 edition, August 1998. Available electronically as `http://www.strandlight.com/EU/files/Manuals/430-550/lp2_2/Tracker.pdf`.

[Thiemann, 1994] Peter Thiemann. *Grundlagen der funktionalen Programmierung.* Teubner Verlag, Stuttgart, 1994.

[van Deursen *et al.*, 2000] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, June 2000.

[VariLiteVirtuoso, 2000] Vari-Lite, Inc., Dalls, Texas. *Virtuoso^{TM} Control Console*, version 2.0 edition, July 2000. Available electronically as `http://www.vari-lite.com/vl/pdf/virt_2_0.pdf`.

[VariLiteVL2400, 2000] Vari-Lite, Inc., Dallas, Texas. *VL2400^{TM} Series Wash Luminaires User's Manual*, August 2000. Available electronically as `http://www.vari-lite.com/vl/vl2000/files/2400user_21.pdf`.

[Wadler *et al.*, 1998] Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language without even being odd. In *1998 ACM-SIGPLAN Workshop on ML*, pages 24–30, Baltimore, Maryland, September 1998.

[Wadler, 1992] Philip L. Wadler. The essence of functional programming. In *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

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

[Walrath and Campione, 1999] Mary Walrath and Mary Campione. *The JFC Swing Tutorial*. Addison-Wesley, 1999.

[Wan and Hudak, 2000] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 242–252, Vancouver, British Columbia, Canada, June 2000. Volume 35(5) of SIGPLAN Notices.

[Wilson, 1992] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. International Workshop on Memory Management, IWMM 92*, pages 1–34, St. Malo, France, September 1992. LNCS 637.

[Winskel, 1993] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[Wirsing, 1990] Martin Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13. Elsevier Science Publishers, Amsterdam, 1990.

Index