Ben-Gurion University of the Negev,
Department of Mathematics & Computer Science

BOOMS
Booms Object Oriented Music System

Thesis submitted as part of the requirements for the
M.Sc. degree of Ben-Gurion University of the Negev

by

Eli Barzilay:

⋆ *Maze is Life!* ⋆

(eli@cs.bgu.ac.il)

The research work for this thesis has been carried out at
Ben-Gurion University of the Negev,
under the direction of Dr. Mira Balaban

December 17 1996

This is the standard place to thank my wife, and apologize for making her life such a mess. I'm no exception: I thank Regina for being. Infinity should be her reward:

$$\lambda x.xx$$

Many more Thank instances should be allocated all around. Many of them will remain implicit, but three should become explicit:

- Mira Balaban my advisor, for many sleepless nights, good intensions and a strong sense of ontology.
- Michael Elhadad, who sent many useful messages and provided callbacks.
- My mother and father for making me who I am. Non of this could happen without their motivation, implementation and maintenance.

A word on sexist language: I know that referring to the 'user' as a male user is not "politically correct". My native language is Hebrew, which is sexist by default (using male as default). English is a bit better (like the word "user" having no sex), but still, there is no sex-less form for 'he', 'his' etc. I've decided against using long combinations like 'his/hers' or the provocatively distractive form 'hers' — I simply use male as default. The intelligent reader can transform all these to female form if he (she!) wishes.

Have lots of fun,
Eli Barzilay:
Maze is Life!

Abstract

This work addresses the problem of providing a computer-based environment to support the music composition process. One of the prominent aspects of the composition process is the importance of structure: for a composer, a music piece is more than its flat score; it is a structured object, and its structure captures the expressive intention of the composer.

Existing music editors do not provide appropriate support for composition: most only address the score editing process, but more profoundly, because there is no uniform or conventional way to denote structure in music pieces, it has been unclear what information a structural editor for music should manipulate. Given this state of affair, composers must remember or add personal annotations to their compositions to remember their structure. As a result, the editor cannot provide support for the manipulation of the music piece as a structured and meaningful object.

This thesis presents the BOOMS editor intended for music composition. BOOMS supports a combination of structural and non-structural editing of music pieces and demonstrates the added power provided by an explicit representation of structure. The main focus of the work has been on:

- Developing a methodology to combine structural editing in non-structural editors;
- Investigating how abstraction mechanisms can help to turn an editing session into a reusable function, which captures the intention of the composer;
- Formally comparing direct abstraction on the music pieces being edited and abstraction on the history of the commands the composer used to build a piece.

The BOOMS system was developed as a general application framework for developing editors with support for a combination of structural and regular editing and support for end-user abstraction as a tool to define reusable functions without programming. The framework is implemented in CLOS (the Common Lisp Object System) and features a sophisticated Windows interface. Instantiating the framework to a specific editing domain is a simple task, due to the clean object-oriented design of the framework.

While the BOOMS framework is generic, it is most effective when instantiated to domains similar to music composition, where a user incrementally

builds a structured object by using a restricted set of commands to combine smaller units into larger ones. The BOOMS framework was instantiated to three domains to illustrate the support an abstraction-enabled structural editor can provide to end users: music composition, arithmetic expressions and editing in a micro-world of colored cubes (GCalc).

In a BOOMS instantiated editor, the end-user interacts with the actual piece being created using simple editor commands. At the same time, a view of the editing session (as an editing graph) is maintained by the editor. The editing graph can be edited by the user to specify his intentions. Abstraction can also be performed on the editing graph, turning a sequence of editing operations into a new reusable function, that can be used either in the simple editor view or in the structural editing graph view.

A feature of the BOOMS framework is that it defines a place where domain-specific knowledge can be introduced in an editor: the node constructors for the domain and the operators for each type in the domain can be encapsulated in well-defined domain libraries and easily integrated in the BOOMS framework. In particular, the BOOMS music domain editor incorporates well-defined knowledge of music notes, intervals, and arithmetics over them. It is a promising area of future work to extend this package to a more comprehensive music knowledge base.

Contents

Chapter 1

Introduction

This work deals with the problems arising when developing computer music environments to support music composition. It investigates the general issues of hierarchical editing and abstraction as an end-user programming technique within the context of a music editor.

The main motivation of the work is that existing music editors are too limited in scope and functionality to effectively support the composition process. The limits derive from two factors: (1) Most music editors so far have been restricted to score editing and therefore only the result of the composition process is stored and represented, but the structure of the piece, the history of its creation and its motivation is lost. (2) No conventional notation has been accepted within the music community to denote the structure of music pieces. The only "universal" notation for music is the flat score encoded in the common music notation. Therefore there is no accepted way to make-up for the lack of structure support in a score editor by adding mark-up tags within the flat score, as is done in text editors.

Accordingly, the objective of this work is to provide a notation to encode structure in music pieces that can be easily used in a direct-manipulation editor. With this objective, the work has evolved into a general investigation of hierarchical editing and the services a hierarchical editor can provide.

The contributions of the thesis are the development of an application framework for hierarchical editors (the BOOMS system), the definition of a general methodology to combine a domain specific editor with hierarchical editing (through the notion of editing graphs and historical editing) and the study of how hierarchical editing facilitates end-user programming through the use of abstraction on editing graphs.

This generic framework has been applied to create an innovative music editor combining hierarchical and flat-form editing and supporting an intuitive form of abstraction definition. The prototype editor is a promising platform to investigate ways to introduce rich music knowledge within music editors.

1.1 The Hierarchy Aspect in Music

Hierarchical structuring is inherent to music conception. As an acoustic phenomenon, music is a stream of sounds, but it is never conceived as such. Hierarchy is central to any music activity — listening, analysis, composition, tutoring, performance, typesetting and computer processing. In listening, the natural structuring performed in the listener's mind, plays a major role in the generation of music expectations [1]. In analysis, a major subject involves structuring of a music piece along multiple dimensions and analyzing the inter-relationships between such simultaneous structurings in harmony, rhythm, melody and dynamics [2, 3, 4, 5]. In composition, hierarchy has a major role in the actual construction of a piece: abstraction and generalization in the bottom-up direction, and instantiation in the top-down direction are essential composition approaches [6].

The need for hierarchy and structuring in the design and application of computer music tools is intimately interlaced in the short history of computer music [7, 8, 9, 10, 11, 12, 13, 14, 15]. Typesetting has a hierarchical aspect as well [16]. In the design of low level protocols like MIDI [17] and Zippy [18] the need for structure is frequently mentioned.

The work on Music Structures that motivated the BOOMS project concentrates on characterization of the ontology of structured music pieces as a semantic domain for a computer music environment. It suggests the view of a music piece as a temporal structure whose components are nested temporal structures — SMP (Structured Music Piece). The hierarchical construction starts from some primitive layer, that includes sounds (defined by attributes such as pitch or timbre) or atomic motives. The multiple structuring of a single piece gives rise to multiple SMPs, all sharing a single *flat music piece* that acts as their extensional normal form.

The structure of an SMP can be syntactically specified by various construction operators. The basic [direct] operator is general **music-concatenation**,

denoted "**@**". The expression

$$\mathbf{@}(MP_0, t, MP)$$

denotes the extension of the music piece denoted by MP, with MP_0, played at time t, relative to MP's beginning. Two useful special case operators are **simultaneous** and **sequential** concatenations, which are roughly defined as:

Simultaneous: $A \mid B = \mathbf{@}(B, \mathbf{start}(A), A)$

Sequential: $A - B = \mathbf{@}(B, \mathbf{end}(A), A)$

Examples of other useful operators are **transpose**, **reverse**, **stretch**, and **merge**.

In a Computer Music Environment based on the Music Structures view, the operators are the construction means (value constructors). They express the compositional intention of the user. Hence, the history of an editing session evaluates to the intended composition, and provides the logical structure of the composition.

1.2 Introducing "BOOMS"

BOOMS is a generic framework to study and develop sophisticated domain-specific editors. The project is motivated by Computer-Music Environment needs. The eventual goal of the project is to develop a music workstation supporting multiple structured representations for a single music piece, following the *Music Structures* representation of [12]. This has led to the development of a *hierarchical editor*, that manipulates hierarchical music structures rather than the flat music score: the first stage of BOOMS Object Oriented Music Structures (abbreviated BOOMS).

The hierarchical structures in BOOMS are represented by an internal DAG (Directed Acyclic Graph) data structure that contains node objects. Leaves in this hierarchy stand for "atomic" music pieces (usually single notes), and internal nodes are music constructors. Playing a piece amounts to evaluating the structured hierarchy (the DAG) — calculating a *flat* value which is the normal form of the structured music piece. All structured views of a music piece share the same flat normal form (*i.e.*, they sound identical).

The hierarchical representation stands for the composer's intentions, and defines an ontological object (an actual music piece) implicitly: evaluation must be used to retrieve this extension.

1.3 Thesis Plan

In an effort to uncover the potential of hierarchical editing to end-users, the BOOMS editor was abstracted into a generic application framework for structural editing by untangling the editor part from the specific music knowledge. The resulting framework is described in detail in Chapter6. It includes the following layers:

1. Domain-independent node (constructor) management: the Node class. These nodes compose the internal hierarchy data structure.

2. Graphic editor for hierarchy manipulations: the Icon class. This class augments the Node class (using inheritance). The user manages objects of this class that are visualized nodes.

3. On top of these two general layers, a package mechanism layer supports implementation of multiple domains. This way, BOOMS can be used for diverse hierarchical domains.

   Three specific domains are implemented — a music domain, a colored cubes graphical domain and an arithmetic domain.

A contribution of the work is the definition of a general methodology to turn a domain-specific editor into a combined hierarchical-direct editor. The approach is based on the view that each command performed by the user in the native editor on domain values is recorded in an editing graph. In the editing graph, each node corresponds to a user command and combines domain values (the parameters of the command) into new complex values. As the editing graph evolves, every argument is itself a node in the previous state of the editing graph. The editing graph therefore encodes the history of an editing session. This methodology is introduced in Chapter 3.

In structured domains like music composition, the editing graph corresponds to the structure of the music piece being constructed. But because the editing process can evolve in random ways, the editing graph can develop into a tangled web. Because the structure of the music piece is of

importance to the user (it captures his intention), the BOOMS approach allows the composer to directly edit the graph. The BOOMS framework thus fulfills two needs: it implicitly defines a language to describe the structure of a music piece by mapping editor commands to structural constructors, and it provides a direct manipulation tool for the resulting structural objects. This is particularly important for music composition, because there is no conventional way to denote structure for music pieces. The application of the BOOMS methodology to different domains is discussed in Chapter 4.

In addition, the BOOMS editing approach, through the reification of the editing process, allows non-programmers to turn editing sessions into new operators. Because editing actions have been turned into objects (the editing graph), it becomes natural to abstract a subgraph of the editing graph into a new node constructor. This is an intuitive way to create user-defined functions. This work presents an in-depth investigation of the abstraction process enabling such definitions, and compares it with direct abstraction on domain values which was proposed as an alternative technique for end-user programming in [19]. This is discussed in Chapter 5.

Chapter 2

Related Work

This work builds on different disciplines and fields. The main topics addressed are: computer music environments and composition environments; the general editing process; abstraction and end-user programmability.

2.1 Computer Music Environments

In this section we investigate two music environment applications that are both advanced and widely used: DMIX and Common Music.

DMIX

DMIX [13] is a system written by Danny Oppenheim in Smalltalk. The main concept of DMIX is an expressive user-interface which operates in multiple dimensions, including many visualization forms like box-graph, score, and text representation. DMIX entails an extensive set of functions, some of them are used via "slapping" — dragging one music piece and dropping on another, performing some interaction between both.

The combination of multi-dimensional representation with slapping allows for an interesting form of abstraction in the following way: a view can represent the pitch of a music piece, another view the rhythm of another piece; by slapping (dropping) the pitch view on the rhythm view, one can obtain a new music piece composed of these pitch and rhythm specifications. In this operation, the rhythm dimension of the second piece has been abstracted into a function that has been applied to the first piece.

Although very impressive, DMIX has some disadvantages. First of all, there is no formal model for DMIX. It was developed as a private tool by one musician, and extended over time to a full application. Comparing DMIX with BOOMS reveals two additional points where BOOMS provides features that are necessary in such music environments:

- There is no hierarchy information in any form (applying an operation on a music piece yields a new one that has no connection to the original).
- Abstraction is limited to slappability, *i.e.*, one can only abstract a single dimension for which a view already exists in the system — new views cannot be created by end-users, and slappable views cannot be combined. End-user programmability or extensibility is therefore impossible.

Common Music

Common Music [20] is a Common Lisp-based system written by Rick Taube. Comparing Common music to other computer-music tools is similar to comparing Common Lisp to other programming languages. Common Music provides a very rich set of of music primitives and types to support algorithmic composition. Many useful features are implemented like streams, containers and algorithmic music.

Compared to other computer-music applications, Common Music is more programmer-oriented than most applications, however, it is attractive enough to make musicians learn how to use it (a surprising point is that musicians actually spend time to learn how infinite streams are used). The containers feature of Common Music supports hierarchy in compositions, however, this is similar to nested procedures: the non-trivial management of containers and the overhead involved prevents using this feature for constructing music like nodes are used in BOOMS (no Common Music user will ever use a container for three notes).

Common Music is very close to Common Lisp, so features like abstractions are handled using Lisp. Another effect of this is the system being very complex, for example, many macros are used all over (working without the reference manual is an impossible task for many users). This approach makes the usage of advanced features like abstractions not applicable for novice users. There is a graphical user interface implemented (Stella), but it is only partial in the sense that many features are not accessible through it.

The Common Music system demonstrates the need for programmability in composition environments. One of the most interesting features of the system is the pervasive use of streams. BOOMS is different from Common Music by emphasizing hierarchical structure usage and making abstraction an intuitive mechanism to support end-user programmability.

2.2 Historical Editing and Structural Editing

The Programmer's Apprentice

The objective of the *Programmer's Apprentice* project [21] is to study how a knowledge-based editor can help automate the tasks of program writing, modification and documentation. One of the main themes of the research is that the editor must encode explicitly more information than is written in the text of the program in order to appropriately assist the programmer. In the *KBEmacs* prototype, this additional knowledge is encoded in the form of *clichés*. A cliché encodes the knowledge shared by the programmer and an external assistant when modifying a piece of code. The following example shows a cliché defined for enumerating a Lisp list:

```
(DEFINE-PLAN CDR-ENUMERATION
    (PRIMARY-ROLES (LIST)
     DESCRIBED-ROLES (LIST)
     COMMENT "enumerates the elements of {the list}")
  (LET* ((LIST {the input list}))
    (LOOP
      (IF ({NULL, the empty} LIST) (RETURN))
      {({CAR, the current} LIST), the output element}
      (SETQ LIST ({CDR, the rest} LIST)))))
```

Definitions of clichés include a body over which parameters are abstracted and, most importantly, a set of annotations that explicitly describe the roles of parameters and constraints over their instantiation. In the above example, the methods to test if a generalized list is empty is abstracted to the annotation `{NULL, the empty}`, comment annotations are also introduced which are later used to automatically generate documentation for the code. Other annotations can be used to introduce constraints on parameters.

This knowledge is used by the editor to provide the following functionality: during program synthesis, the programmer can select a cliché from

a library and instantiate it using explicit editor commands. The programmer can alternatively enter code directly and an analyzer parses the code to recognize instances of existing clichés. In both cases, the editor maintains an explicit representation of the cliché structure of the code — called the *program plan* — in addition to the program text. Because the program plan is explicitly maintained, the KBEmacs editor can support modification of the program at a much higher level of abstraction than a character based editor can. Beyond parse tree editing (as provided by syntactic editors), KBEmacs claims to support editing at the level of "algorithmic structure". This translates into sophisticated support for program modification: if a programmer decides to switch a CDR-enumeration through a list into an ARRAY-enumeration, the editor can transpose automatically the instances of the CDR-enumeration, and propagate this change to all relevant occurrences.

The *Programmer's Apprentice* project illustrates the need to maintain extra information in addition to the flat domain values being edited, to describe the intention of the designer. The need for such meta-editing knowledge is made even more pressing in the domain of music composition, because of the lack of established mark-up standards in music notation.

In KBEmacs, the integration of the domain value editing and knowledge editing is through a stage of automatic analysis and plan recognition of the user actions. While this approach is conceivable in the programming domain, where a large cliché library can be designed building on extensive results in software engineering and programming language semantics, it is much harder to apply in the music domain, where even a notation for structure is missing, and the notion of "composer intention" is much harder to grasp. In addition, music creators often consciously seek ambiguity in their composition, and a plan recognition mechanism would perform poorly in such conditions.

The alternative approach implemented in BOOMS, is to provide explicit editing of the editing graph, to empower the composer with the possibility to specify his intention. Automatic analysis of the editing actions is beyond the scope of this work.

Besides this difference, in BOOMS as in KBEmacs, the ideal place to introduce domain specific knowledge to introduce sophisticated services in the editor is in the set of editor commands. The BOOMS music knowledge base is encapsulated in a library of editor commands, appearing to the user in a palette, and plays a role parallel to KBEmacs's cliché library.

Chimera

In the *Chimera* graphical editor [22], Kurlander investigated ways to automate repetitive tasks in user interfaces. The *Chimera* system is built using demonstrational techniques — empowering users to "program an application through its user interface." Kurlander developed five powerful techniques to automate repetitions. Most relevant to this work are the techniques of *editable graphical histories* and *macros by example* built on top of them.

Graphical histories encode in a comic strip metaphor the commands used in an editing session: "Commands are distributed over a set of panels that show the graphical state of the interface changing over time. These histories use the same visual language as the interface" ([22, p.11]). The graphical history is automatically maintained as the user performs actions on the editor. Strategies are designed to make histories shorter and focus on significant actions in the system. Declarative rules encoding regular expressions of commands are used to analyze the stream of commands issued by the user and coalesce similar commands into a single pane of the history.

Histories serve as a basis for a sophisticated form of undo-redo where the user can select which section in the history to undo or redo, and have the editor propagate changes through the rest of the session as required. In addition, histories are used as the basis for a form of abstraction implemented in the *graphical macro by example* technique of *Chimera*. The main concept is to abstract histories into functions by generalizing some of the objects manipulated in a sub-session. The abstracted sub-session is then named and can be used as a new command.

The BOOMS approach is heavily influenced by Kurlander's work, in its focus over histories (which are called editing graphs above). The main difference in approach is that *Chimera* depicts histories in the same visual language as the object editor, while BOOMS depicts them in a hierarchical view, focusing on their structural properties. The latter approach is a design decision motivated by the need to denote structure in music composition. In BOOMS, the differences between the properties of the editing graph and the values being edited is also investigated formally.

2.3 Abstraction and End-user Programmability

Orlarey and others in [19] propose to build a music calculus based on pure lambda calculus. To illustrate the use of abstraction, they develop an editor in a micro-world of colored cubes, that can be combined around the three dimensions. The main idea investigated in their system is to provide the user with the possibility to abstract over the primitive values being edited. The resulting domain includes both the colored cube values and the functions obtained as a result of abstraction.

This system is discussed at length in Chapter 4, where a reimplementation is presented (the GCalc system), together with a formal analysis of the domain and of the abstraction process. The BOOMS approach mainly differs from the GCalc model in that abstracted functions are not part of the value domain. This difference is parallel to that with the *Chimera* system — in BOOMS, both editing graphs and abstracted functions are objects that the composer wants to represent and manipulate. The BOOMS hierarchical view provides a language to denote structure when no such formalism is generally available in music.

Chapter 3

Hierarchical Editing — BOOMS

In this chapter we introduce the notion of hierarchical and historical editing. We show that editing sessions themselves, can be characterized as structured expressions. Hierarchical editing is the manipulation of such structured expressions. It is contrasted to flat-form editing, where the result of the editor operation is always fully evaluated: hierarchical editing is intensional, while flat-form editing is extensional. At last, we introduce historical editing, as a dual editing mode, where the user can manipulate both the flat-form of his creation and edit directly the operational hierarchy of his editing session. BOOMS supports a restricted form of historical editing that we call "Constructive Historical Editing".

3.1 Editing as a Structured Process

In this section, we develop the view of an editing session as a structured process, represented by some hierarchical expression. This is enabled by viewing an editor as generating a functional domain: an editor is always in a state which is represented by a value (or values), and the user can perform operations which bring the editor to other states. The values representing the editor state are the functional domain values, and the user-operations are functions on these values.

A simple example for an editor and a corresponding domain is a pocket-calculator. It has a state which is composed of several number values — at

least two numbers are needed for using the simple binary functions, and there is usually some kind of a memory storage place. The arithmetic operations are functions that operate on the calculator's current values and set new ones. These values and the user functions attached to the calculator keys compose a functional domain.

We proceed to representing an editing session as an expression hierarchy. This is immediate once having a functional representation for an editor domain, since functional expressions are naturally viewed as trees (*e.g.*, syntax trees). Given a set of values V (editor states) and a set of functions F (editor operations), let H denote the set of hierarchical expressions over $\langle V, F \rangle$. The value of a hierarchical expression is given by a function Eval : $H \mapsto V$ that evaluates expressions in H into the actual values in V — by applying the corresponding functions in F.

Note that the tree representation of functional expressions is appropriate when they are pure values. If hierarchy values are objects (with identity) then the representation uses DAGs (Directed Acyclic Graph).

3.2 Editing Graphs are Intensional

Editing is an operation along time. An editor that keeps track of editing operations along some period of time, and allow editing of the corresponding operational hierarchy, is called a *Historical Editor*. Such an editor generates a functional term for each operation. These terms preserve the history; their evaluation yields the intended result. The generation of this hierarchical expression is domain-independent. It is natural for a historical editor to have two views: one for the original [extensional value] domain and support direct manipulation, and another for the hierarchical expressions that represent the processes that created these extensional values.

In this section, we first compare the domain of values manipulated in the flat form editor (the original domain of values) and the values manipulated in the historical editor. As discussed in Chapter 2, it is often claimed that these two domains should be kept as close as possible, to allow the user to manipulate histories in the same way as domain values. We develop a formal analysis of editing graphs to precisely explain how editing graphs differ from domain values. In the next section, we analyze a second reason why the domain of editing graphs and the value domain differ: certain editing commands are destructive, and leave no trace in the value being constructed.

3.2.1 Intensionality and Hierarchy Editing

The critical difference between domain values and editing graphs is that the later are intensional. Indeed, a historical editor enables direct manipulation of the operational hierarchy. The BOOMS editor supports graphical DAGs editing. This way, using subexpressions of the final creation is an inherent feature; logical subunits of our creation can be directly referenced. Another advantage is the support of multiple top-level hierarchies (multiple roots), rather than a single value.

When editing, the handling of objects (values with identity) is more appropriate than pure values. That is, object manipulation is descriptively more powerful than manipulation of values. For example, the expression "2×2" can use two different 2's, or the same single 2 object. These two options represent different intentions — like calculating the area of a 2-by-2 rectangle, or the area of a square with an edge length of 2. We also gain the benefit of better evaluation times, since each node in the graph is evaluated only once. This brings us closer to the expression graphs in [23].

In the intensional approach, the hierarchical operation expressions are DAGs: nodes represent operations, and edges represent the relationship of a subexpression node being an argument of a parent expression node. Manipulating an acyclic graph rather than a tree means that subexpression-sharing is allowed. Henceforth, the term "historical editing" is always used in its intensional meaning (manipulation of objects).

3.2.2 Formal Description

To precisely explain the intensional nature of the historical editing domain, we propose a formal description of the process of maintaining the editing graph over a flat form editor.

The Functional Domain

The functional domain $D = \langle V, F \rangle$ is a pair of the values set, and the functions set, as follows:

- V is any set of values.
- F is a set of functions over V:
  $F \subseteq V^n \mapsto V$

The Hierarchical World

The definition of the hierarchical world $W = \langle H, C, \text{Eval} \rangle$, based on the functional domain D, is a triplet of hierarchical values, constructor functions and an evaluation function, as follows:

- C has a constructor function $\tilde{f}$ for each domain function f in F (n is used as the function's arity, $\tilde{f}$ has the same arity):
  $\forall f \in F : \exists \tilde{f} \in C :$
  $\tilde{f} : H \mapsto H$
  $\forall x_1, x_2, \ldots, x_n \in H :$
  $\tilde{f}(x_1, x_2, \ldots, x_n) = X \iff$
  $X = \tilde{f}(y_1, y_2, \ldots, y_n) \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge \ldots \wedge x_n = y_n$

- H is defined using primitive values V and augmented by the constructors in C:
  - $V \subseteq H$
  - $x_1, x_2, \ldots, x_n \in H,\ \tilde{f} \in C \Longrightarrow \tilde{f}(x_1, x_2, \ldots, x_n) \in H$
  - No other values are in H.

- The Eval function gives the semantic value of the syntax values in H:
  $\text{Eval} : H \mapsto V$
  $$\text{Eval}(x) = \begin{cases} \text{x} & \text{if } x \in V \\ f(\text{Eval}(x_1), \text{Eval}(x_2), \ldots, \text{Eval}(x_n)) & \\ & \text{if } x = \tilde{f}(x_1, x_2, \ldots, x_n),\ \tilde{f} \in C \end{cases}$$

3.3 The Editing Graph Preserves Destructive Operations

The general operation set of an editor can be categorized into two classes:

Constructive operations These are the usual operations — operations that make us "advance toward the final goal". Most of the 'interesting' operations fall in this category, they are 'interesting' because they add information — they create explicit combinations: applying operator O on states $S_1, \ldots, S_n$ is simply $O(S_1, \ldots, S_n)$ (a construction or a value).

There are many examples for constructive operations, they are usually domain dependent because they construct values (or compute a value)

specific to the domain of the values they operate on. Text writing / pasting in a text editor and value entering / calculating on a calculator are constructive operations.

Destructive operations These are operations that make the editor go back to some previous values (or state) like the 'undo' operation, or produce values that are not explicit combinations of previous ones, like the 'delete' operation.

Note that this usage of the term 'destructive' is different from its use in programming languages — changing the internal state of an object.

These operations, contrasted to constructive operations, tend to be domain independent for reasons explained below. Examples of destructive operations are deletion, undoing and resetting.

Surely, constructive operations should be recorded in the historical structures created by the editing session — otherwise we will not be able to remember the way a value was produced.

On the other hand, destructive operations are used to recover previous editing state / values, so we want this feature to be reflected in the corresponding hierarchical values created. This means that destructive operations are not implemented as constructors for hierarchy values, but as functions that map hierarchy values to other hierarchy values — these can be considered as meta-level [destructive] operations for hierarchy maintenance.

There can be a situation where the user wants to keep the destructive operations in the hierarchy — like when abstracting a sequence of operations using their hierarchical form. Think of keyboard-macros in Emacs which carry a similar kind of abstraction — it can be the sequence of '`A`'–DEL–'`B`', which is exactly equal to '`B`' (and the destructive information is redundant), but it can be the sequence DEL–'`A`' where the destructive operation must be part of the macro.

3.3.1 Example: Usage Scenario

This example demonstrates destructive operations. We use a different editor domain — a text editor. Text is a stream of characters with no structure and the operations used to create it define the operational history structure. The only operations are concR for concatenating on the right, and ConcL on

the left. The text string '`ABC`' can be the result of

$$\text{concR}(\text{concR}(\text{`A'}, \text{`B'}), \text{`C'})$$

or

$$\text{concL}(\text{concR}(\text{`B'}, \text{`C'}), \text{`A'})$$

These two options for creating the '`ABC`' text correspond to two possible ways for a user to generate this text — typing '`A`', '`B`' and '`C`', or typing '`B`', '`C`' and then '`A`' at the beginning.

If this editor allows destructive operations like undo, we can have more history expressions that generate this text, like:

$$\text{concR}(\text{undo}(\text{concR}(\text{concR}(\text{`A'}, \text{`B'}), \text{`D'})), \text{`C'})$$

which happens when the user mistakingly enters '`ABD`', then undoes the '`D`' and finally adds the correct '`C`'.

3.3.2 Constructive Editing

The implementation of selective historical editing by using meta-level operations (partial evaluation of destructive editing operations) for destructive operations is called *Constructive Editing Approach*. For example, in the above text editor domain, when the user corrects the mistaken '`D`' character, we get a corresponding subexpression of the form $\text{undo}(\text{concR}(X, Y))$. Such expressions can easily be identified and replaced by their simpler equivalent form X — this simpler expression is the result of a meta-level delete operation.

If the constructive editing approach is to be implemented in a hierarchical editor, then the user should be given a way to say whether he wants such automatic reductions or not, as there are cases (mentioned above) where the destructive information should be kept.

Note: the BOOMS system does not handle editor domains that include destructive operations. It allows, however, manual meta-level operations like deletion on hierarchy structures.

3.4 Conclusion: Historical Editing

This chapter has developed a view of editing as a constructive process, where the editor automatically maintains an editing graph reflecting the actions

performed by the user on a flat form editor. This editing graph can then be directly edited by the user. We have precisely highlighted the differences between the domain of values manipulated by the flat form editor and those manipulated by the history editor: editing graphs are intensional even when the value domain is extensional, and editing graphs record destructive operations, and, therefore, dealing with destructive operations in a historical editor requires special care.

Hierarchical Editing of the operational history has the following advantages:

- Can describe any history of editing, on any editor domain.

  As we've seen — any editing session can be represented by an operational hierarchy, so we do not lose any editing power. The price can be a loss in user-interface clarity for certain domains. This is discussed in Chapter 4.

- Enable direct manipulation of the hierarchy using logical units of the creation.

  When representing the history hierarchy of a normal editing session, we will probably get something which is of little use. The possibility of working directly with the hierarchy allows the creation of expressions that are more meaningful.

  For example, using a text editor example, writing the string '`ONE TWO`' in the obvious way (from left to right) creates the expression:

  $$((((((\text{`O'} \cdot \text{`N'}) \cdot \text{`E'}) \cdot \text{`\ '}) \cdot \text{`T'}) \cdot \text{`W'}) \cdot \text{`O'})$$

  Hierarchical editing can produce a more meaningful representation:

  $$((((\text{`O'} \cdot \text{`N'}) \cdot \text{`E'}) \cdot \text{`\ '}) \cdot ((\text{`T'} \cdot \text{`W'}) \cdot \text{`O'}))$$

  The subexpressions '`ONE`' and '`TWO`', can be used elsewhere (using the intensionality feature).

- The intensional nature enables simultaneous editing of multiple hierarchies (the hierarchy is a DAG with possible several roots). For example, the calculator from the above example can have several memory cells.

- The intensional visualization of the hierarchy editor enables direct access to subexpressions, which encourages reusing of expressions. This way a user can build true *sharing* of subexpressions, and distinguish it from *repetition*, as in the above example of "2×2". This distinction is meaningless in a flat editor, which always expands (computes) its values. A historical editor recording the operational hierarchy of a flat-editor, can, for example, use shared subexpressions using the copy / paste operations — remembering that the two subexpressions are actually the same one — and then enforce propagation of changes to its two instances.

- The creation process can proceed in all possible design directions — top-down, bottom-up and combinations.

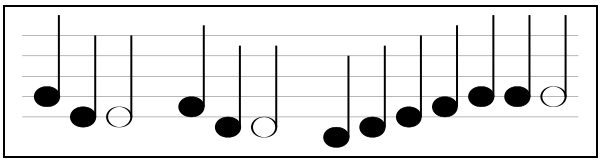
  Working bottom-up is obviously possible, this is what is created when representing a 'flat' editing session (the possibility of holding multiple unrelated expressions allows the building of low-level expressions, and then higher expressions). Using unspecified arguments, it is also possible to start with the top level function expressions, working down to more specific expressions — thus supporting top-down method. Any combination of these methods, like starting from the middle level, down to the bottom and then up to the top, is, of course, possible.

  This is an obvious feature for any DAG editor, but here we have achieved this advantage for any editor domain.

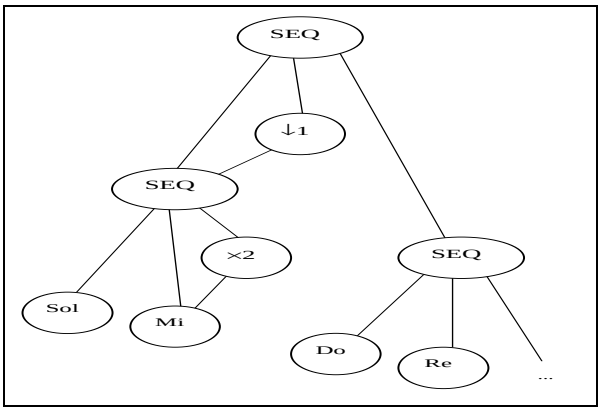
Chapter 4

The BOOMS Methodology on Different Domains

The previous chapter focused on the notion of historical editing. In this approach editing means manipulating the hierarchical structures created by user operations. For example, as the user constructs the following music piece:



the following editing graph is constructed:



We also discussed the differences between the value domain (the score) and the editing graph domain.

This chapter illustrates these differences in greater details by focusing on three examples: simple arithmetics; a music composition domain; the colored cube example introduced by Orlarey and others at the Grame computer-music center in Lyon, France in [19].

These three samples demonstrates three different possible scenarii for an editor implemented in BOOMS:

1. A native domain can have no structure at all. This is demonstrated by the arithmetic sample, where the native values are numbers.

2. The native domain can have an intended structure that is not represented by the editing operation graph. This is the case of editing music pieces using a flat editor: the usual way for entering a music piece is by sequentially specifying its elements — the corresponding graph is not meaningful, see page 19.

3. Finally, the structure of domain values can be described using a BNF, and the editor commands correspond exactly to the non-terminals of the BNF. This is the case of GCalc, where there are few ways to construct a given cube. In this case the editing graph will be almost isomorphic to the value being constructed. Still, the editing graph of a value, and its structure can differ, since the editing graph is an intensional DAG.

GCalc is the main example that is discussed in this chapter. The GCalc editor is used as an example of a flat-form editor. GCalc was designed and implemented for the purpose of "investigating radically new approaches to end-user programming." We reimplemented this system in Scheme to first learn on ways to integrate abstraction in user interface and to precisely understand the differences between the BOOMS historical editing approach and the GCalc "flat domain" approach. In this chapter, the GCalc domain and editor operations are presented. Abstraction is postponed to the next chapter, where it is confronted with the structure abstraction operation in BOOMS.

Example

We first illustrate the approach of viewing an editing session as a hierarchical expression in the arithmetic domain[1].

We take $\mathbb{N}$ as the functional domain, with the simple arithmetic functions: $\{+, -, \times\}$. The hierarchical world H is created, based on the functional domain:

1. The set of values, V, is taken as the natural numbers, $\mathbb{N}$. This is the set of H's primitive values: $V = \mathbb{N} \subseteq H$.

2. For each $f \in \{+, -, \times\}$, we create a binary constructor function (with the same name) over H — these functions create *new* values in H.

3. The evaluation function maps values in H to values in V in the obvious way.

We compare now the values in V (the original set of values), and the values in H:

- The functional domain, V, contains only the natural numbers — $\mathbb{N}$. The result of applying a function on these values is another numeric value, so the three applications "$2 + 4$", "$8 - 2$" and "2×3" produce the *same* value — "6".

- H also contains the naturals as its primitive values, but now $\{+, -, \times\}$ are value constructors. So, for example, "$2 + 4$", "$8 - 2$", "2×3" and "6" are all *different* values.

- The evaluation method gives the intuitive mapping between H and V — "Eval$(2 - 4)$", "Eval$(8 - 2)$" and "Eval(2×2)" produce the same result — "6". (Writing Eval$[\![2 + 4]\!]$ might be more appropriate as the values of H are actually syntax values).

This is similar to the expression graphs discussed in [23]. A graph can be viewed as a hierarchy value, and graph reduction as an implementation of an evaluation function (when producing a primitive-value result).

[1]This example is implemented in BOOMS as the 'arithmetic' demo package.

Historical Editing and the Music Domain

Section 1.1 describes the world of Structured Music Pieces. This can be the hierarchical representation of a 'flat' editor that manipulates music pieces as note streams (similar to a textual editor handling character strings).

The Music Structures domain provides a natural example for editing as a hierarchical expression. This is so because the main concatenation operators are true value constructors. A concatenation of two structured music pieces evaluates to a new *composed* value, that explicitly includes the original pieces. This holds also when concatenating flat music pieces, although the resulting value is shallow-structured (concatenation of primitive elements). The evaluation (using Eval) of a concatenated value yields a flat music piece that explicitly contains the same primitive elements.

This is different from the status of values in the arithmetic example due to the evaluation function. For example, the expression "2×3" is evaluated to the number "6", and the original expression is lost.

4.1 GCalc Domain Description

The purpose of GCalc is to demonstrate how to build a programming language on top of a structured world via abstraction and investigate abstractions as means for conceptualization. The GCalc domain is simple, structured and can be supported by off-the-shelf graphic visualization tool. It is better suited than a straightforward textual domain. The implementations are simple but powerful, and can produce surprisingly complex results.

The data values of GCalc are colored cubes. An informal description follows (a formal description is given in Section 4.3):

- There are primitive cubes which are uniformly colored with a single color. These can be colored with any color, and a transparent color is usually very handy.

- There are three ways of combining cubes to generate compound ones — Left-Right, Top-Bottom and Front-Back. Each of these is a binary constructor. The visualization of a compound cube is always a construction of two cubes (using the constructor's axis) of equal size — that is, the two sub-cubes are shrunk in half to fit their part of the compound cube.

- There are definitions for an abstraction and an application operations. This is the most important aspect of GCalc. It is discussed rigorously later in Chapter 5 (Section 5.2).

Note that cubes in GCalc are data values and not objects. This means that each cube value is unique, for example, there are no two different red cubes. Cubes should be thought of as cube-concepts being conceptually constructed, and not as physical cubes.

4.2 The GCalc Flat-Form Editor

The implementation of the GCalc editor is an example for a simple and intuitive user-interface. It is a flat-form interface for the structured domain of cubes. A brief description of the interface of the Scheme implementation (which is similar to the Grame implementation) follows:

The workspace is divided into three panes[2] — the *work area*, the *primitive colors palette* and the *storage*. Each pane is a grid of cells. Almost all operations are performed by drag-and-drop between the panes cells. Utility operations like naming a cube, saving and loading, are not discussed (the full documentation for the Scheme implementation can be found in Appendix C).

The Primitive Colors Palette

The grid of primitive colors palette contains cells with the primitive cubes. It is used as a supply of basic cubes for dragging. An extra operation performed on this pane is abstraction.

The Storage

The storage is a grid of empty cells that can be used for storing intermediate results. Constructed objects can be dragged to and from the storage. This is a necessity as construction is bottom-up (thus some way for saving objects must exist or else operations can only be performed on the current cube and a primitive cube).

[2]See page 28 for a screen snapshot

The Work Area

The work area pane is the most important one — it is the application main window. This is the place where the "current cube" is being worked on, the place where all of the work is done. It is a grid of 3×3 cells as follows:

Back	Top	Apply (func)
Left	Selected	Right
Apply (arg)	Bottom	Front

Selected This is the selected cell that holds the cube the user is currently working on. This cube can be dragged to any storage cell for later use, and any cube from the storage pane or the primitive colors palette can be dragged here for more editing.

Note that this cube can be dragged like any other cube, so it can also be placed in one of the operation cells described below.

Left This is the cell that generates Left-Right constructions. When a cube is dragged here, a Left-Right construction of this cube and the selected cube is created and the result is placed in the selected cube.

Right This is the opposite cell for the "Left" cell, it will create a Left-Right construction of the selected cube and the dragged cube (same as above, but with the arguments swapped), and place the result in the selected cube.

Top This creates a Top-Bottom cube, similar to the "Left" cell.

Bottom The opposite of "Top".

Front This creates a Front-Back cube, similar to the "Left" cell.

Back The opposite of "Front".

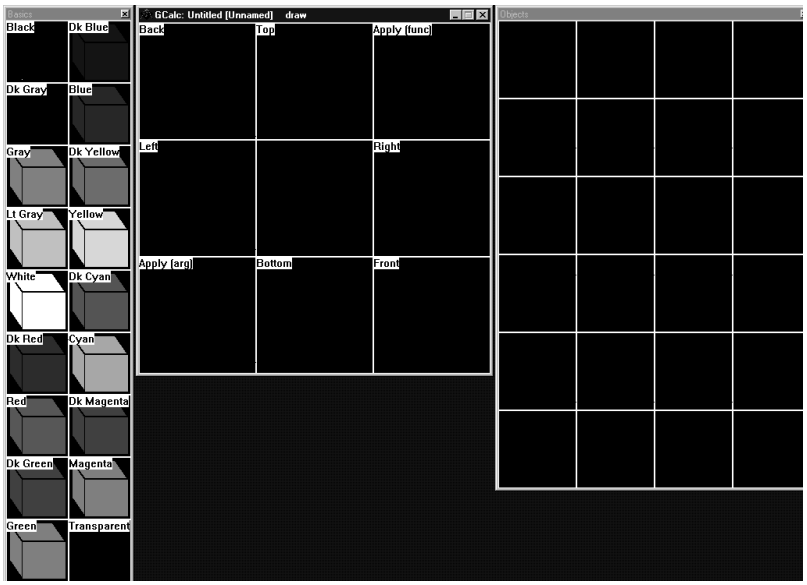
Apply (arg) / Apply (func) The Abstraction and Application are the important operations of the GCalc system. They are discussed later, but the user interface is briefly described here.

Creating an abstraction can be done only on a basic colored cube on the primitive colors palette, by right-clicking on the color to abstract.

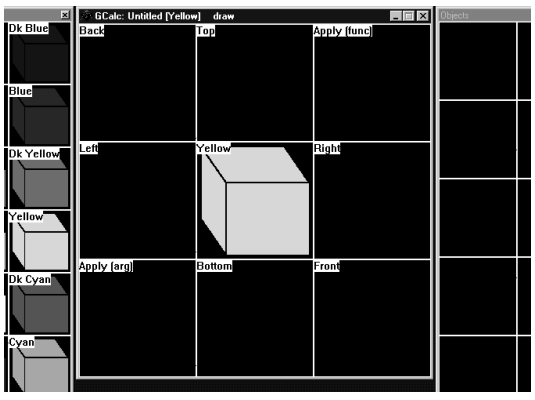
Applying an abstraction is done by dragging an object to one of the "Apply" cells: the "Apply (arg)" is for dropping argument values for a function in the selected cell, and the "Apply (func)" is for dropping a function to be applied on an argument in the selected cell.

Session Sample

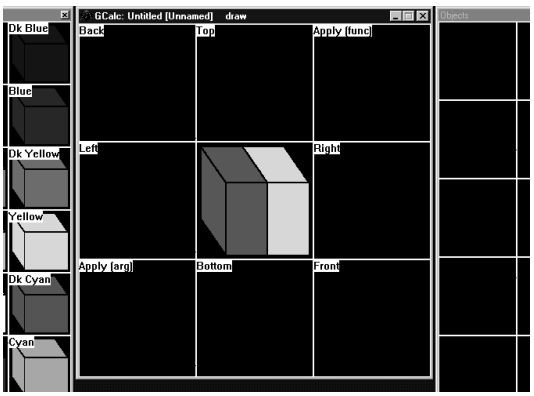
The session sample given here is taken from a running session of the Scheme implementation. No abstractions are used in this sample.



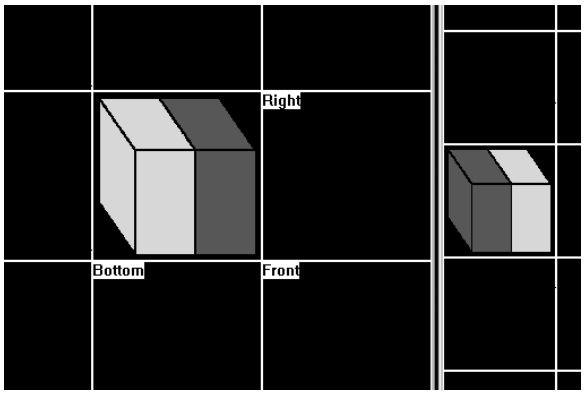
The initial GCalc screen and floating windows



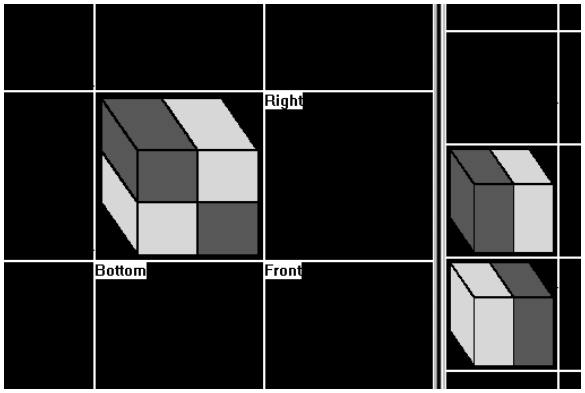
Dropping an initial yellow cube



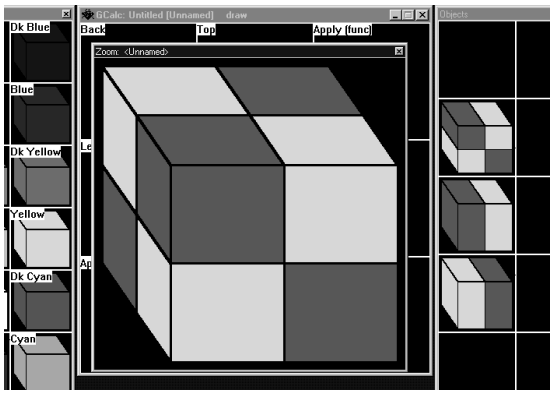
Dropping a red box on its left



Save the result and build the mirror-image object



Saving this object as well, and put the two one on top of the other



Save that, and finally: construct another object and put it behind. View the final result on the zoom pane.

4.3 GCalc's Formal Definition, Part I: COLORS

The GCalc domain is formally defined in three parts. Part I is given here, and Parts II–III that deal with abstractions, are described in Sections 5.2.1 and 5.2.3. The whole definition can be found in the documentation section of the GCalc Scheme file.

The first part defines the simple colored cube ADT world — named "COLORS". These are the definitions for the values, and the similarity equivalence relation which defines the visualization properties — similar objects looks identical (although they may actually be composed differently), and behave identically.

4.3.1 Values

The GCalc ontology (real-world elements) is the intended semantics behind the syntax. This is defined first to motivate an intuitive approach to the rest of the description. The ontology section contains the values and constructors given here, and selectors that are defined in the next section.

1. C — an infinite set of colors and color constructs and a special "*undef*" value which is an absorbing value for all functions.

2. B — a non-empty subset of C,
   the subset of primitive basic elements (color concepts).

3. *Cons* — a set of constructors which are functions $C \times C \mapsto C - B$ defined by:
   $Cons = \{\text{LeftRight}, \text{TopBottom}, \text{FrontBack}\}$
   $\forall F \in Cons:$ F is a strict constructor:
   $\forall x_1, x_2 \in C:$
   $x_1, x_2 \in C - \{undef\} \Longrightarrow$
   $F(x_1, x_2) = X \Longleftrightarrow X = F(y_1, y_2) \wedge x_1 = y_1 \wedge x_2 = y_2$
   $x_1 = undef \vee x_2 = undef \Longrightarrow$
   $F(x_1, x_2) = undef$

4. C is composed of its basic core B, and extended by the constructors in *Cons*:

(a) $undef \in C$

(b) $c \in B \Longrightarrow c \in C$

(c) $c_1, c_2 \in C, F \in Cons \Longrightarrow F(c_1, c_2) \in C$

(d) No other values are in C

4.3.2 Selectors

Six selectors are defined — two for each constructor. For example, the first selector defined for a LeftRight construction is LeftPart:

$\text{LeftPart} : C \mapsto C$

$$\text{LeftPart}(c) = \begin{cases} \text{x} & \text{if } c = \text{LeftRight}(x, y) \\ \text{TopBottom}(\text{LeftPart}(x), \text{LeftPart}(y)) & \\ & \text{if } c = \text{TopBottom}(x, y) \\ \text{FrontBack}(\text{LeftPart}(x), \text{LeftPart}(y)) & \\ & \text{if } c = \text{FrontBack}(x, y) \\ undef & \text{if } c \in B \text{ or } c = undef \end{cases}$$

This should be done for all six selectors. The process is generalized by two selector generators — higher order functions Select1 and Select2 defined as follows:

$\text{Select1}, \text{Select2} : Cons \mapsto (C \mapsto C \cup \{undef\})$

$\forall F \in Cons, c \in C :$

$$\text{Select1}(F)(c) = \begin{cases} \text{x} & \text{if } c = G(x, y) \wedge F = G \\ G(\text{Select1}(F)(x), \text{Select1}(F)(y)) & \\ & \text{if } c = G(x, y) \wedge F \neq G \\ undef & \text{if } c = undef \vee c \in B \end{cases}$$

$$\text{Select2}(F)(c) = \begin{cases} \text{y} & \text{if } c = G(x, y) \wedge F = G \\ G(\text{Select2}(F)(x), \text{Select2}(F)(y)) & \\ & \text{if } c = G(x, y) \wedge F \neq G \\ undef & \text{if } c = undef \vee c \in B \end{cases}$$

Call this property: Distribution of the selectors over the constructors in Cons.

Naming convention — if the above F's name is AaaBbb, then we use these names for the selectors:

- Select1(AaaBbb) — AaaPart
- Select2(AaaBbb) — BbbPart

so the six selectors we have now are: LeftPart, RightPart, TopPart, BottomPart, FrontPart and BackPart.

Examples:

- TopPart(TopBottom(LeftRight(*Red*, *Blue*), LeftRight(*Blue*, *Red*))) = LeftRight(*Red*, *Blue*)
- LeftPart(TopBottom(LeftRight(*Red*, *Blue*), LeftRight(*Blue*, *Red*))) = TopBottom(*Red*, *Blue*)
- LeftPart(TopBottom(LeftRight(*Red*, *Blue*), *Green*)) = *undef*

4.3.3 Similarity

The GCalc ontology is intensional in the sense that its values stands for equivalence classes over a similarity relation ($\approx$). This relation defines the visualization ("real-world" implementation) properties. As a general principle, all values in the same similarity equivalence set will look identical (although they may actually be composed differently), and behave identically for all functional combinations. The intuition for this is the existence of values that are built differently but look the same and are indistinguishable, like

$$\text{TopBottom}(\text{LeftRight}(X, Y), \text{LeftRight}(Y, X))$$

and

$$\text{LeftRight}(\text{TopBottom}(X, Y), \text{TopBottom}(Y, X))$$

Note, however, that "X" is not similar (and certainly not equal) to the construction "LeftRight(X, X)", so these two values are visualized differently.

The similarity equivalence relation $\approx$ is defined as follows:

$\approx : C \times C$ such that

$$\begin{aligned}
\approx = \; & \{\langle b_1, b_2\rangle \in B \times B \mid b_1 = b_2\} \cup \\
& \{\langle c_1, c_2\rangle \in (C - B) \times (C - B) \mid \\
& \quad c_1 = F(x, y) \wedge \\
& \quad \text{Select1}(F)(c_1) \approx \text{Select1}(F)(c_2) \wedge \\
& \quad \text{Select2}(F)(c_1) \approx \text{Select2}(F)(c_2)\}
\end{aligned}$$

(Note that $\langle \mathit{undef}, \mathit{undef}\rangle \notin \approx$.)

Examples:

- $\text{TopBottom}(\text{LeftRight}(\textit{Red}, \textit{Blue}), \text{LeftRight}(\textit{Green}, \textit{Yellow})) \approx$
  $\text{LeftRight}(\text{TopBottom}(\textit{Red}, \textit{Green}),$
  $\text{TopBottom}(\textit{Blue}, \textit{Yellow}))$

- $\text{LeftRight}(\textit{Red}, \textit{Red}) \not\approx \textit{Red}$

[We can define an extensional ontology — in terms of equivalence classes over the similarity relation and find a normal form for all values. This is not done here since it is more complex, and we want this ontology to represent the actual implementation.]

4.3.4 Syntax

The GCalc syntax is a context free grammar, that is used to represent the ontological values. We use capitals for syntactic categories and lower case for terminals. The syntax is equivalent to the ontology, therefore no semantic function is defined — it should be simple (*e.g.*, the syntax "LeftRight(*Red*,*Blue*)" means the result of applying the LeftRight constructor function on the *Red* and *Blue* values).

1. COLOR $\longrightarrow$ BASIC | COMPOSED

2. BASIC $\longrightarrow$ 'red' | 'green' | 'blue' | 'yellow' | 'white' | 'black' | ... |
   'transparent'

3. COMPOSED $\longrightarrow$ 'LeftRight' (COLOR , COLOR) |
   'TopBottom' (COLOR , COLOR) |
   'FrontBack' (COLOR , COLOR)

Note that different symbols derived from BASIC require different objects to be contained in B, this restriction is implied by the semantics of GCalc. In the Scheme implementation, the basic colored cubes are simply represented by a number composed of three values for RGB, except for the transparent cube which is represented by a symbol. So the above syntax is a somewhat simplified version of the actual rule used:

BASIC $\longrightarrow \mathbb{N}$ | 'transparent'

4.4 Implementing GCalc in BOOMS

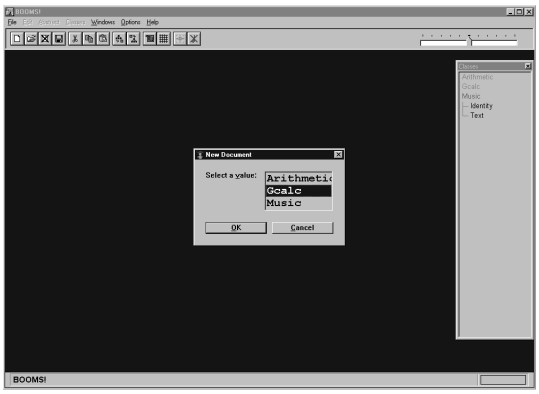
The GCalc domain is structured by a few primitive values and three binary constructors. In BOOMS, the three binary constructor operations are implemented as operation nodes and the basic colors as leaf nodes. For each binary constructor, its dual operation (the right-left operation is the dual operation of a left-right construction) is implemented by a node that applies the constructor on its swapped arguments.

BOOMS is an intensional editor, which manipulates objects rather than values. Hence, the implementation of GCalc as a package of BOOMS treats cubes as objects — it allows two different red cubes. In this respect, the BOOMS implementation differs from the flat-form GCalc editor. Obviously, the BOOMS implementation of GCalc lacks the extensional user-interface.

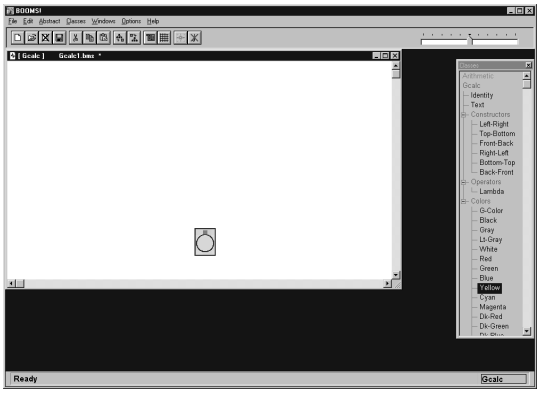
When actually using the BOOMS hierarchy editor to play with cube objects, it is clear that this interface is better for organizing cube ideas and playing with them, *i.e.*, testing various possibilities. Nevertheless, it is equally clear that the flat-form editor provides a more intuitive, direct user-interface. The flat-form editor allows *working in the domain* while the hierarchy editor forces a hierarchical abstraction on the creative process. The visual representation of nodes, is the only link between the BOOMS editor and the concrete GCalc domain.

4.4.1 Session Sample

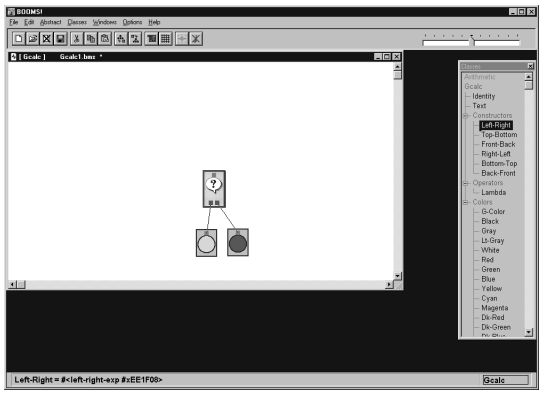
The session sample given here is taken from a running session of the BOOMS implementation. No abstractions are used in this sample. This example is actually identical to the one in Page 28. Comparing the two sessions it is clear that the flat-form native editor is much more intuitive.



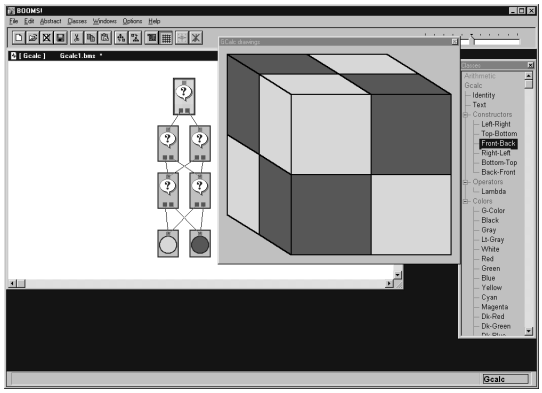
Starting BOOMS — selecting a document type to create



Dropping a yellow leaf constant from the classes palette



Building the right-left construction



The final object, viewed in the window that pops on evaluation

Note that this working method naturally supports top-down construction which is impossible in the flat-form editor.

Chapter 5

Abstractions

Up to now we used only simple domains, all functions were built-in functions defined in the domain of the editor. In this chapter we discuss user-definable functions — abstractions. Two types of abstractions are discussed — flat-form abstractions which are performed on a 'flat' value, and structure abstractions which are performed on intensional hierarchy expressions.

5.1 Introducing Abstraction & Application

What are abstraction and application? A brief explanation is given here, for an excellent introduction of this idea see [19].

Abstraction is an act of generalization which is done on two values, the exact types of values used may vary, but usually we abstract a composed value on some simple value. Conceptually, abstracting a composed value v on some simple value x means "stripping off" the x property from v, creating some general object — a function to be applied later. For example, abstracting the red color from a red flower gives a generalized, colorless flower. Technically, the result of such an abstraction, is the value v with each occurrence of x replaced by a variable — a hole for later filling.

Application is the opposite operation — instantiating an abstraction. It takes an abstraction and some appropriate value, and instantiates (replace) the abstraction's variable by the applied value. The obvious semantics for an abstraction is a function, and application can be described as the result of applying this function. For example, application of the above colorless flower on a yellow color yields a yellow flower.

5.1.1 The Power of Abstraction

In an editor, abstractions are usually very useful. In fact, some complex objects have an explicit representation which is so complex, that abstractions are necessary for an understandable description. Having abstractions in an editor is powerful because it means we have the power of a programming language. An example of the power one gets having an editor capable of abstracting can be found in [22]. The abstraction and application operations are simple enough that they can be merged into any domain (specifically — any editor domain), this is one of the main ideas of [19] (demonstrated by GCalc).

BOOMS implements flat-form and hierarchy abstractions. Flat-form abstraction is used to abstract some value which is a component of a flat-form creation. Hierarchy abstraction is more like the keyboard macro facility in Emacs — abstracting on the operational hierarchy means we record some operations to be used on other values later. These abstractions are performed on the operation DAG, so they can select the variable part of the abstraction intensionally.

5.2 Flat-Form Abstractions

Flat-form abstraction is a direct implementation of lambda abstraction (see [23, Chapter 2] for a description of lambda calculus). The flat values of the domain are compositions of simple values; the composite values abstracted on the simple values. This is done by replacing all occurrences of a simple value in the composite one by a variable, producing an abstraction value. Thus we need new objects: *abstractions* and *variables*.

Abstractions are created using a constructor that is given a variable and an object, this means that we also need a function that will do the actual replacement of the abstracted value by a variable.

Variables are an infinite set of values distinct from the domain values. This set should be infinite since a new abstraction uses a new variable for avoiding the problem of name capturing.

As an example, take the character concatenation domain from Section 3.3.1. The flat-form domain is the strings domain, with values that are character

concatenation. No hierarchy structure is used, the string '`ABC`' is simply a concatenation of '`A`', '`B`' and '`C`'. The value of 'Twice' in

$$Twice := \text{Abstract}(\text{`A'}, \text{`AA'})$$

is an abstraction (a function) that takes some value (a string value) as an argument, and returns this value repeated twice:

$$\text{Apply}(Twice, \text{`ABC'}) = \text{`ABCABC'}$$

Abstraction is possible on another abstraction to create multiple argument functions (the arguments are 'processed' one-by-one, this is called "Currying" — [23, Section 2.1.1]):

$$ABBA := \text{Abstract}(\text{`B'}, \text{Abstract}(\text{`A'}, \text{`ABBA'}))$$

$$\text{Apply}(\text{Apply}(ABBA, \text{`ABC'}), \text{`ABD'}) = \text{`ABDABCABCABD'}$$

As seen in this example, the abstraction and application operations are similar to search and replace operations performed separately, with the addition of variables and scoping rules.

The final part of this discussion handles visualization of abstraction values. When writing an abstraction on paper it is easy to use something like

$$ABBA = \lambda\beta.\ \lambda\alpha.\ \alpha\beta\beta\alpha$$

which is conventional. When visualizing such an object, the obvious (and intuitive) way for visualizing this is using the domain operations with "variable objects" placed in the abstracted value. The text example makes the usage of special characters ('α' and 'β') a possible solution based on the correspondence between simple values and variables ('β' replacing B). The visualization method used in GCalc solves this by using a variable set that have variables corresponding to simple values (and therefore can be visualize as variations of such). A simple variable set V based on the simple value domain $\mathbb{N}$ can be $V = S \times \mathbb{N}$, using this, and the representation '$\overline{\texttt{X}}$', '$\overline{\overline{\texttt{X}}}$', ... as the variables corresponding to the character '`X`', then we have the above $ABBA$ represented as:

$$ABBA = \overline{\texttt{B}}.\overline{\texttt{A}}.\ \text{`}\ \overline{\texttt{A}}\ \overline{\texttt{B}}\ \overline{\texttt{B}}\ \overline{\texttt{A}}\ \text{'}$$

(The initial variable declaration is still needed for identifying an abstraction value as well as specifying the variables and their order.)

5.2.1 GCalc's Formal Definition, Part II: FCOLORS

This is the functional extension for COLORS (compare with the first part in Section 4.3). It extends the COLORS world with lambda abstraction and application objects, and an Eval function which implements the reduction rules. It contains various functions that are used for handling the lambda calculus.

5.2.1.1 Values

1. C — is extended with abstraction and application values generated by the new constructors and the variables defined below.

2. B — is the same as B in COLORS.

3. $Vars = B \times N$ — is an infinite set of variables that derive from the set B. This means that each variable is associated with some basic color in B, this is because we want to keep the abstraction variable's color when visualizing it. We use a short form — Red_0 instead of $\langle Red, 0\rangle$. Equality of variables is defined as:
   $\forall x_i, y_j \in Vars:$
   $x_i = y_j \iff x = y \wedge i = j$

   For example, Red_0 and Red_1 are two different variables which derive from the *Red* color. In general we have infinitely many variables with an attached color.

4. $BV = B \cup Vars$ — the set of simple (non composed) values.

5. *Cons* — is the same as in COLORS.

6. *FCons* — the two lambda-calculus constructors — a set of strict (see *Cons* above) functions in $C \times C \mapsto C - BV$ is defined by:
   $FCons = \{\text{Abstraction}, \text{Application}\}$

   These constructors are used by the Lambda and Apply functions described below to generate Abstraction and Application values.

   Note: the approach of having some value for an abstraction rather then building a value was taken here because of the GCalc environment which handles abstraction values like all other values.

7. $CCons = Cons \cup FCons$ — the set of all constructors.

8. C is composed of its basic core B, and extended by the constructors in $CCons$:

   (a) $undef \in C$

   (b) $c \in BV \Longrightarrow c \in C$

   (c) $c_1, c_2 \in C, F \in CCons \Longrightarrow F(c_1, c_2) \in C$

   (d) No other objects are in C

5.2.1.2 Selectors

All previous selectors are maintained. There is no distribution over the *FCons* constructors. All COLORS selectors evaluate to undef for variables and Abstraction / Application values:
$\text{Select1}, \text{Select2} : Cons \mapsto (C \mapsto C \cup \{undef\})$
$\forall F \in Cons, c \in C$:

$$\text{Select1}(F)(c) = \begin{cases} \text{x} & \text{if } c = G(x,y) \wedge F = G \\ G(\text{Select1}(F)(x), \text{Select1}(F)(y)) & \\ & \text{if } c = G(x,y) \wedge F \neq G \wedge G \in Cons \\ undef & \text{if } c = undef \vee c \in BV \vee \\ & \quad (c = G(x,y) \wedge G \in FCons) \end{cases}$$

$$\text{Select2}(F)(c) = \begin{cases} \text{y} & \text{if } c = G(x,y) \wedge F = G \\ G(\text{Select2}(F)(x), \text{Select2}(F)(y)) & \\ & \text{if } c = G(x,y) \wedge F \neq G \wedge G \in Cons \\ undef & \text{if } c = undef \vee c \in BV \vee \\ & \quad (c = G(x,y) \wedge G \in FCons) \end{cases}$$

For example:

- $\text{LeftPart}(\text{Abstraction}(Red_0, \text{LeftRight}(Red_0, Green))) = undef$

5.2.1.3 User Ontology Functions

These functions are used to construct abstractions and perform applications. The auxiliary functions used here are defined in the next section, this includes the Eval function that defines the semantics of abstraction and application.

1. **Lambda function:**
   This function creates abstraction values (in analogy to `lambda` forms that create closure values).

   Lambda : $B \times C \mapsto C$
   $\forall b \in B, c \in C$:
   $\text{Lambda}(b, c) = \text{Abstraction}(b_i, \text{Replace}(c, b, b_i))$
   where $b_i \in \mathit{Vars}$ does not occur in c (that is, $\neg\text{Occur}(c, b_i)$ — we can choose the minimal i).

   For example:

   - $\text{Lambda}(\mathit{Red}, \text{LeftRight}(\mathit{Red}, \mathit{Green})) =$
     $\text{Abstraction}(\mathit{Red}_0, \text{LeftRight}(\mathit{Red}_0, \mathit{Green}))$

   [See Occur and Replace function definitions below.]

2. **Apply function:**

   This function is a mere wrapper around the Eval function which evaluates applicable functions — does the beta-reductions.

   Apply : $C \times C \mapsto C$
   $\forall c_1, c_2 \in C$:
   $\text{Apply}(c_1, c_2) = \text{Eval}(\text{Application}(c_1, c_2))$

   For example:

   - $\text{Apply}(\text{Abstraction}(\mathit{Red}_0, \text{LeftRight}(\mathit{Red}_0, \mathit{Green})), \mathit{Blue}) =$
     $\text{Eval}(\text{Application}($
     $\text{Abstraction}(\mathit{Red}_0, \text{LeftRight}(\mathit{Red}_0, \mathit{Green})),$
     $\mathit{Blue})) =$
     $\text{LeftRight}(\mathit{Blue}, \mathit{Green})$
   - $\text{Apply}(\mathit{Red}, \mathit{Green}) =$
     $\text{Application}(\mathit{Red}, \mathit{Green})$

   [See the Eval definition below.]

5.2.1.4 Auxiliary Functions

This section defines functions used for the ontology functions and the evaluation semantic function.

1. **Occur function:**

   This function checks if its second argument occurs (is a sub value) of the first argument.

   $\text{Occur} : C \times BV \mapsto \mathit{Bool}$
   $\forall c \in C, b \in BV :$

   $$\text{Occur}(c, b) = \begin{cases} \mathit{True} & \text{if } c = b \\ \text{Occur}(c_1, b) \vee \text{Occur}(c_2, b) & \\ \qquad \text{if } c \neq b \wedge c = F(c_1, c_2) \wedge F \in \mathit{CCons} & \\ \mathit{False} & \text{if } c \neq b \wedge c \in BV \end{cases}$$

   For example:

   - $\text{Occur}(\text{Abstraction}(\mathit{Red}_0, \text{LeftRight}(\mathit{Red}_0, \mathit{Green})), \mathit{Red}) =$ False
   - $\text{Occur}(\text{Abstraction}(\mathit{Red}_0, \text{LeftRight}(\mathit{Red}_0, \mathit{Green})), \mathit{Red}_1) =$ False
   - $\text{Occur}(\text{Abstraction}(\mathit{Red}_0, \text{LeftRight}(\mathit{Red}_0, \mathit{Green})), \mathit{Red}_0) =$ True

2. **Replace function:**

   This function replaces all occurrences of the second argument in the first argument by the third argument.

   $\text{Replace} : C \times BV \times C \mapsto C$
   $\forall c \in C, b \in BV, x \in C :$

   $$\text{Replace}(c, b, x) = \begin{cases} \text{x} & \text{if } c = b \\ F(\text{Replace}(c_1, b, x), \text{Replace}(c_2, b, x)) & \\ \qquad \text{if } c \neq x_1 \wedge c = F(c_1, c_2) \wedge F \in \mathit{CCons} & \\ \text{c} & \text{if } c \neq x_1 \wedge c \in BV \end{cases} \quad \lhd$$

   For example:

   - $\text{Replace}(\text{LeftRight}(\mathit{Red}_0, \mathit{Green}), \mathit{Red}, \mathit{Blue}) =$ $\text{LeftRight}(\mathit{Red}_0, \mathit{Green})$
   - $\text{Replace}(\text{LeftRight}(\mathit{Red}_0, \mathit{Green}), \mathit{Red}_0, \mathit{Blue}) =$ $\text{LeftRight}(\mathit{Blue}, \mathit{Green})$
   - $\text{Replace}(\text{LeftRight}(\mathit{Red}, \mathit{Green}), \mathit{Red}, \mathit{Red}_0) =$ $\text{LeftRight}(\mathit{Red}_0, \mathit{Green})$

3. **Eval function:**

   This function evaluates expressions. It is applied immediately on application constructions, as implemented by the system. This can be modified, however, for example to obtain a system with a functional meaning to an application of a basic colored cube. It is defined using an algorithm — the reduction rules algorithm:

```
Eval : C --> C
Function Eval(c)
  if c in BV
    return c
  else if c = Abstraction(v_i,b)
    return c
  else if c = Application(c1,c2)
    let x1 = Eval(c1), x2 = Eval(c2)
      if x1 in BV or x1 = Application(y1,y2)
        return Application(x1,x2)
      else if x1 = Abstraction(v_i,b)
        return Eval(Replace(b,v_i,x2))
      else
        return Application(x1,x2)
  else
    c is F(c1,c2) for some F in Cons
    return F(Eval(c1),Eval(c2))
```

   For example:

   - Eval(LeftRight(*Red*, *Green*)) = LeftRight(*Red*, *Green*)
   - Eval(Application(LeftRight(*Red*, *Green*), *Red*)) =
     Application(LeftRight(*Red*, *Green*), *Red*)
   - Eval(Application(
     Abstraction(Red_0, TopBottom(Red_0, *Green*)),
     Blue)) =
     TopBottom(*Blue*, *Green*)

5.2.1.5 Similarity

The FCOLORS similarity is an extension of the COLORS similarity. The new similarities are:

- Variables are treated like B values.
- An Application value is similar to another Application with similar components.
- An Abstraction value is similar to another Abstraction with variable based on the same color, and similar body when using the same variable. Note that this is not functional equivalence.

The new similarity equivalence relation $\approx$ is redefined as follows:
$\approx : C \times C$ such that
$\approx =$
$\{\langle b_1, b_2\rangle \in B \times B \mid b_1 = b_2\} \cup$
$\{\langle v_i, v_j\rangle \in \mathit{Vars} \times \mathit{Vars} \mid u_i = v_j\} \cup$
$\{\langle c_1, c_2\rangle \in (C - BV) \times (C - BV) \mid$
$\quad c_1 = F(x, y) \wedge F \in \mathit{Cons} \wedge$
$\quad \text{Select1}(F)(c_1) \approx \text{Select1}(F)(c_2) \wedge$
$\quad \text{Select2}(F)(c_1) \approx \text{Select2}(F)(c_2)\} \cup$
$\{\langle c_1, c_2\rangle \in (C - BV) \times (C - BV) \mid$
$\quad c_1 = \text{Abstraction}(u_i, b_1) \wedge c_2 = \text{Abstraction}(v_j, b_2) \wedge$
$\quad u_i, v_j \in \mathit{Vars} \wedge u = v \wedge$
$\quad \text{Replace}(b_1, u_i, w_k) \approx \text{Replace}(b_2, v_j, w_k) \wedge$
$\quad\quad$ for some $w_k \in \mathit{Vars}$ such that
$\quad\quad\quad \neg(\text{Occur}(b_1, w_k) \vee \text{Occur}(b_2, w_k))\} \cup$
$\{\langle c_1, c_2\rangle \in (C - BV) \times (C - BV) \mid$
$\quad c_1 = \text{Application}(x_1, y_1) \wedge c_2 = \text{Application}(x_2, y_2) \wedge$
$\quad x_1 \approx x_2 \wedge y_1 \approx y_2\}$
For example:

- Application(LeftRight(*Green*, *Red*),
  TopBottom(LeftRight(*Red*, *Blue*),
  LeftRight(*Green*, *Yellow*))) $\approx$
  Application(LeftRight(*Green*, *Red*),
  LeftRight(TopBottom(*Red*, *Green*),
  TopBottom(*Blue*, *Yellow*)))

- Abstraction(Red_0, Application(Red_0, *Green*)) ≈
  Abstraction(Red_1, Application(Red_1, *Green*))

5.2.1.6 Syntax

1. COLOR ⟶ BASIC | COMPOSED
2. BASIC ⟶ ‘red’ | ‘green’ | ‘blue’ | ‘yellow’ | ‘white’ | ‘black’ | . . . |
   ‘transparent’
3. COMPOSED ⟶ ‘LeftRight’ (COLOR , COLOR) |
   ‘TopBottom’ (COLOR , COLOR) |
   ‘FrontBack’ (COLOR , COLOR) |
   ‘Lambda’ (BASIC , COLOR) |
   ‘Apply’ (COLOR , COLOR)

Note that Abstraction and Application are not included in the syntax — Lambda and Apply are used instead.

5.2.2 Generalizing GCalc's Abstractions

The flat-form abstraction defined for GCalc is generalized in two directions — making it generally applicable for any editor domain and having the abstraction values (objects) behave like the editor domain native values.

Flat-form Abstractions for Any Domain

The GCalc flat-form abstractions demonstrate a powerful feature that can be implemented in other domains as well. This is used in BOOMS for creating a general program interface that implements flat-form abstractions when provided with simple search and replace operations (these operations depend on the domain simple data values and compound values).

It is implemented in the arithmetic and the music packages, the arithmetic implementation is awkward because the arithmetic expressions are used for the abstraction, while the flat-form is actually a number. The GCalc package is the only one with a visualization for abstractions, the other packages do not implement a flat-form editor so there is no need for such visualization.

Flat-Form Abstractions as Domain Native Values

The goal of having a functional domain on top of the native domain is achieved by merging the functional objects into the domain: making them behave like domain values. This is further emphasized by an obvious question a user asks when playing with GCalc: "What will happen if I'll attach two abstractions?" An answer to this question specifies how abstraction values interact with the host domain.

Merging flat-form abstractions into the domain is an inherently domain dependent job — what needs to be specified is the semantic meaning to constructions of abstraction values. The GCalc specification is natural: a left-right construction of two functions is a function that when applied gives the result that is a left-right construction of the left function on the left half of the argument and the right function on the right half (this is defined in Section 5.2.3 below).
The music and the arithmetic packages in BOOMS do not handle flat-form abstraction constructions. This can be easily added to the music package, following the guidelines of [19] — having a "Music Lambda Calculus" implementation.

5.2.3 GCalc's Formal Definition, Part III: GCALC

This is the final stage, where we have only the extension of FCOLORS to handle functional meaning of constructions of functions.

All that is done here is adding the T-Selectors and the redefinition of the Eval function to handle Applications of Abstraction constructions. The intuition for this extended evaluation rules are activating part of the operator on corresponding parts of the operand.

The Values, Ontology Functions, Similarity and Syntax versions are the same as the FCOLORS definitions. They are part of this definition, but not rewritten here.

5.2.3.1 Selectors

All FCOLORS selectors are also GCALC selectors. For each FCOLORS selector Sel, there is a new GCALC selector TSel that gives the same results as Sel except for BV values, which TSel returns the argument opposed to Sel

which returns *undef*. The intuition is that these selectors will always return the requested part, 'splitting' the argument when needed.

The TSelect1 and TSelect2 are higher order functions similar to Select1 and Select2:
$\text{TSelect1}, \text{TSelect2} : \mathit{Cons} \mapsto (C \mapsto C \cup \{\mathit{undef}\})$
$\forall F \in \mathit{Cons}, c \in C :$

$$\text{TSelect1}(F)(c) = \begin{cases} \text{x} & \text{if } c = G(x,y) \wedge F = G \\ G(\text{TSelect1}(F)(x), \text{TSelect1}(F)(y)) & \\ & \text{if } c = G(x,y) \wedge F \neq G \wedge G \in \mathit{Cons} \\ \text{c} & \text{if } c \in BV \\ \mathit{undef} & \text{if } c = \mathit{undef} \vee \\ & \quad (c = G(x,y) \wedge G \in \mathit{FCons}) \end{cases}$$

$$\text{TSelect2}(F)(c) = \begin{cases} \text{y} & \text{if } c = G(x,y) \wedge F = G \\ G(\text{TSelect2}(F)(x), \text{TSelect2}(F)(y)) & \\ & \text{if } c = G(x,y) \wedge F \neq G \wedge G \in \mathit{Cons} \\ \text{c} & \text{if } c \in BV \\ \mathit{undef} & \text{if } c = \mathit{undef} \vee \\ & \quad (c = G(x,y) \wedge G \in \mathit{FCons}) \end{cases}$$

The names of the generated selectors are the same as those generated by Select1 and Select2, prefixed with a "T".
For example:

- $\text{LeftPart}(\text{TopBottom}(\text{LeftRight}(A,B), \text{LeftRight}(C,D))) = \text{TopBottom}(A,C)$
- $\text{TLeftPart}(\text{TopBottom}(\text{LeftRight}(A,B), \text{LeftRight}(C,D))) = \text{TopBottom}(A,C)$
- $\text{LeftPart}(\text{TopBottom}(\text{LeftRight}(A,B), C)) = \mathit{undef}$
- $\text{TLeftPart}(\text{TopBottom}(\text{LeftRight}(A,B), C)) = \text{TopBottom}(A,C)$

5.2.3.2 Functions

Same as FCOLORS auxiliary functions, except that the Eval function is redefined to handle applications of abstraction constructions.

- **Eval function:**

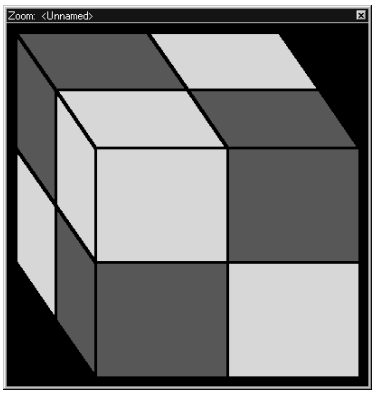
This function evaluates expressions. It is defined using an algorithm — the reduction rules algorithm:

```
Eval : C --> C
Function Eval(c)
  if c in BV
    return c
  else if c = Abstraction(v_i,b)
    return c
  else if c = Application(c1,c2)
    let x1 = Eval(c1), x2 = Eval(c2)
      if x1 in BV or x1 = Application(y1,y2)
        return Application(x1,x2)
      else if x1 = Abstraction(v_i,b)
        return Eval(Replace(b,v_i,x2))
      else
        x1 = F(s1,s2)
        return
          F( Eval(Application(s1,TSelect1(F)(x2))),
             Eval(Application(s1,TSelect2(F)(x2))))
  else
    c is F(c1,c2) for some F in Cons
    return F(Eval(c1),Eval(c2))
```

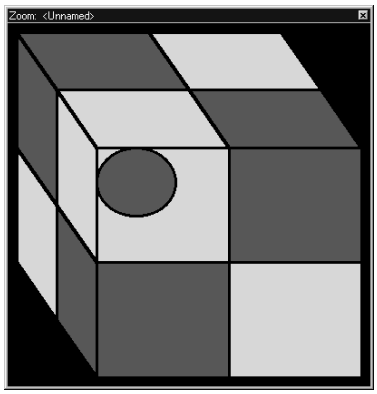
For example:

– Eval(Application(LeftRight(*Red*, *Green*), *Red*)) =
LeftRight(Application(*Red*, *Red*), Application(*Green*, *Red*))

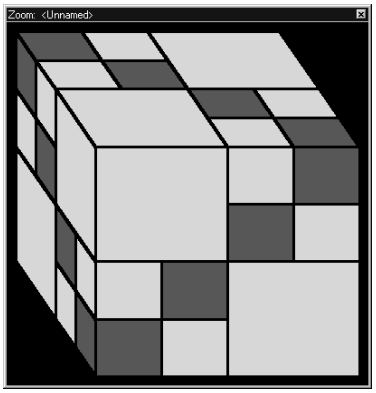
5.2.4 Session Sample



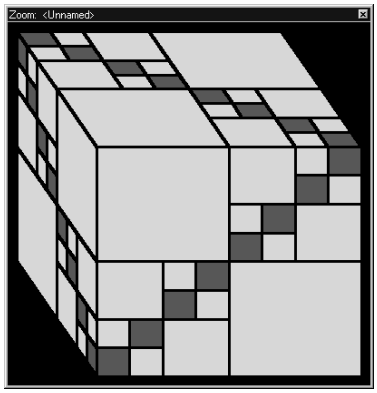
We have seen the object from previous examples, call it "A"



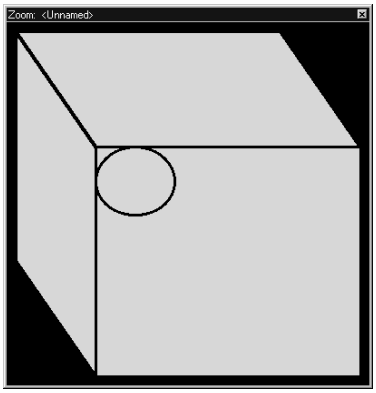
It is now abstracted on the red color, call this function "B"



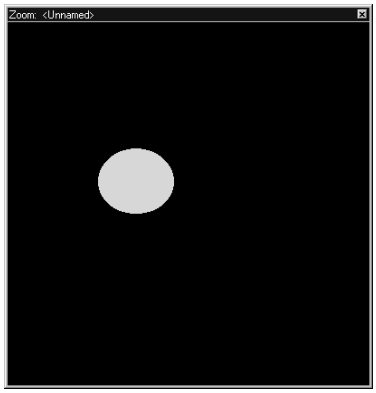
This is the result of applying B to A



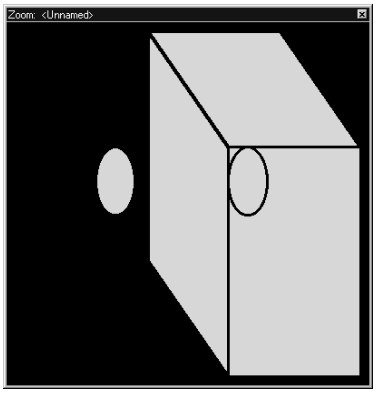
And again — applying B to the last result, name this "c"



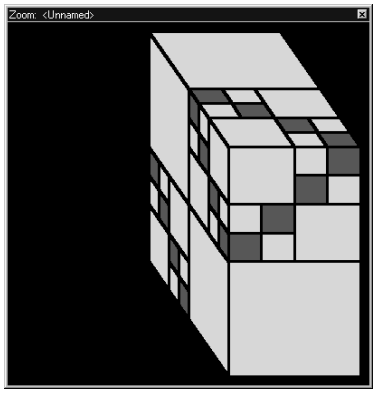
This is the identity function — $(\lambda(Y)\ Y)$



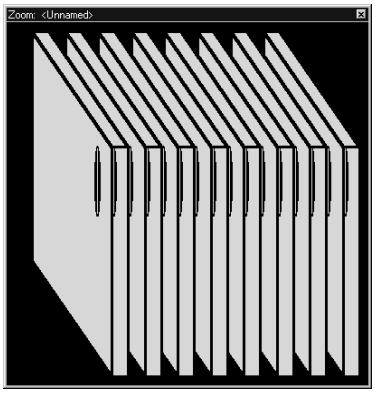
And this is a constant function that will always return an invisible cube — a "hole"



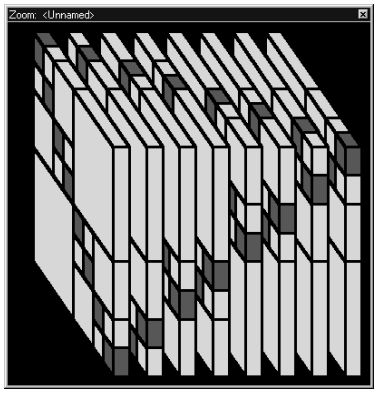
When this functional composition is applied on an object, its right half remains the same, but the left one disappears



This is the result of applying this on C



This one is called a “slicer”



And this is really nice!
(176 simple constructors should be used to achieve this with no abstractions)

5.2.5 Flat-Form Abstractions in BOOMS

Flat-form abstraction is an intuitive operation, that can be conceived as being *part of the domain* rather than being an *operation performed on the domain*. Indeed, this is how flat-form abstractions are implemented in the BOOMS environment. The two operations of abstraction and application become part of the operational hierarchy, since they are treated as domain operations.

5.3 Structure Abstractions

Using flat-form abstractions we can only abstract on the explicit form of the value. For example, given the arithmetic hierarchical expression "2×3+2×4" one cannot abstract only on the first or on the second "2" above, or on the "2 × 3" subexpression. One must abstract on a primitive value, and the abstraction applies to all occurrences of that value. Structure abstractions can distinguish between occurrences of the abstracted value or be applied to complex subexpressions.

A structure abstraction is an abstraction on a hierarchical intensional expression. The process can be described roughly as selecting a subexpression of an expression, and making it the abstraction variable. When such an abstraction is applied on an expression argument, this expression is replaced for the abstraction variable. Generally speaking, structure abstractions are viewed as meta-operations, abstracting edit operations — providing similar power to that of macro languages found in many modern applications.

Structure abstraction is like visually drawing a box around a pattern of operator applications thereby turning it to a compound function. This means we can still use the flat-form abstraction / application machinery for structure abstraction, except for the equality test marked with a '◁' in the Replace function definition on page 45 above. This should compare objects rather than values (use `eq` instead of `equal`). For example, if the DAG representing the expression "2 × 3 + 2 × 4" contains two different "2" leaf objects, then each one can be abstracted alone. Another option is abstraction on the whole "2 × 3" object: each subexpression is itself an object.

A formal definition of structural abstraction and application can be defined similarly to the addition of flat-form abstraction on top of GCalc.

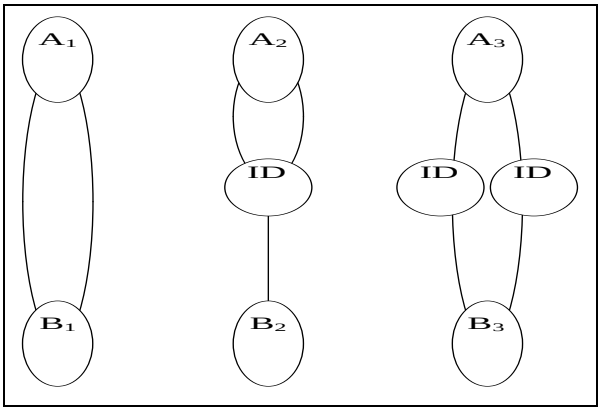
5.3.1 Structure Abstraction in BOOMS

Structure abstraction can be implemented the same way as the flat-form abstraction implementation — have an 'Abstract' and an 'Apply' operations that are merged into the domain, becoming part of the operational hierarchy. Implementing structure abstraction in the same way will be very simple, the main change is using the Lisp '`eq`' function instead of '`equal`' (`(eq x y)` is true only if `x` and `y` are the *same* object while `equal` is used to compare *values*).

However, as already said, we view structure abstraction as meta-level operations. This means we don't have such operations *in* the domain, we rather have these operations work *on* the hierarchical structure — the structure abstraction mechanism is therefore different from the flat-form abstraction implementation.

The most important difference is that a structure abstraction results in a new operation type that can be used — this makes structure abstraction a tool for creating user defined functions (that encapsulate a sub-session). Since abstractions create new operation types, they are abstracted on any number of variables at once. The abstraction object cannot be used for further abstraction, *i.e.*, Currying (the method used for multiple argument flat-form abstraction) is impossible.

Another difference from flat-form abstraction appears when defining what happens if we abstract on an object that has multiple occurrences of the same object as arguments. Using the search operation with object equality testing, we get a single variable that replaces all occurrences. Such behavior is sometimes unwanted, since the intensional argument specification may be expected to behave as if there are no subexpression repetition. The solution to this problem is having a user option that specifies the desired behavior. In any case, an identity node is provided to allow for both behaviors, this is demonstrated by the following diagram:



1. The first abstraction, created by abstracting A_1 alone, can have two variables, for the two A_1 arguments or one variable for the single B_1 object. This should be resolved by the user option setting.

2. The second abstraction, created by abstracting A_2 and *ID*, will always have one argument — there is a single external link to a single object.

3. The third abstraction, created by abstracting A_3 alone, will always have two arguments — there are two external links to two different *ID* objects.

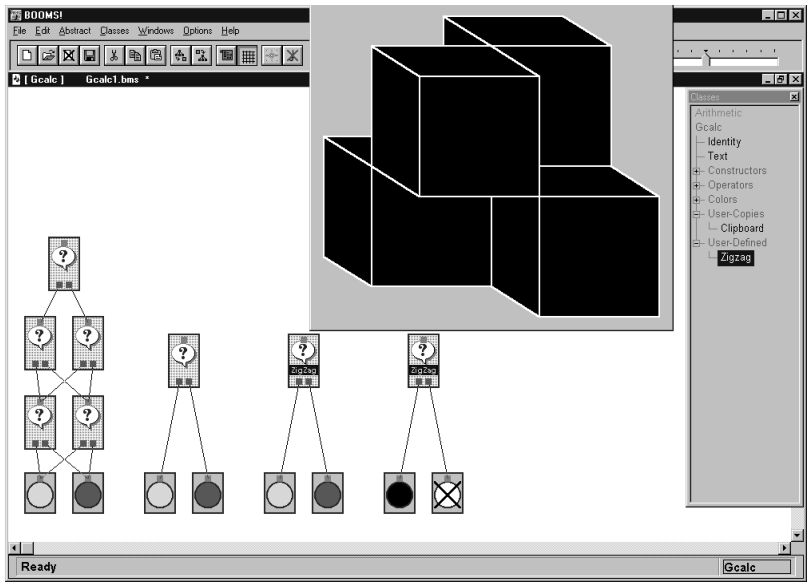
Note: there is a limitation on the abstracted expression — its subgraph may be connected to other nodes only as arguments to the subgraph leaves or the root being an argument of other nodes. Other connections are prohibited since abstractions are used for zooming out (replacing the subgraph by an abstraction node), and there is no intuitive solution to visualizing such links. See the next section for more details.

5.3.2 Hiding Details

Using a hierarchical editor, it is usually desired to have a way for "zooming out" unwanted details. This means the user can hide parts of the hierarchy that are not currently relevant to the editing process. For example, a composer might want to design the overall structure of a music piece after creating all of its basic parts, or improving a basic music component that is already placed in the global composition. This approach of zooming-out is different from geometrical approaches, where the zooming is achieved using graphical representation (for example — [24]).

The essence of such a "zoom out" operation is similar to abstracting. In both cases details are hidden — in the zoomed out expression, or in the abstraction body. With this similarity in mind, the structure abstraction mechanism was designed to provide also zooming-out capability. This is achieved by allowing the creation of unnamed abstractions. In that case the abstraction is immediately instantiated (a node is created), and it replaces the original abstracted expression, hiding its details inside the abstraction body. This eliminates the need for two separate mechanisms: the BOOMS user interface does not mention any "zoom out" option, only unnamed abstractions.

5.3.3 Session Sample



From left to right:

1. This is from the previous example on page 4.4.1, we want to abstract the selected icons (all but the bottom two).
2. The abstraction has been performed — this is the zoom-out operation.
3. Now it is given a name — the name also appears on the classes palette. This is learning a new user operation.
4. Finally, we apply this new node on black and transparent.

Mixing both abstraction methods can lead to confusing situations — like Lambda(*Red.Red* | *Green*) and [structure] abstract this on the *Red* cube: applying this on *Green* creates a different function. This happens because structure abstractions are done *on* the hierarchy, and flat-form abstractions are *part of* the hierarchy.

Chapter 6

System Outline

BOOMS provides a general framework for hierarchical editing. This includes an intensional DAG data structure with node elements, a built-in structure editor that handles this structure, and hierarchical abstraction facility. The system supports multiple editing domains having a separate package implementation for each, three sample packages are provided. All interface functions are specialized in the domain package, allowing multiple editing interface implementation — for example, having a flat-form view alongside the structure view.

BOOMS is more appreciated by programmers than by end-users. The main goal is to develop an extendible system that can serve as a good basis for more editing domains on one hand, and an elaborate user interface on the other. The system is large[1], therefore only the guidelines principles are presented here — see the Appendix for a complete documentation and program listings.

6.1 Nodes

Nodes are the building blocks of the intensional DAG structure. Each node represents some subexpression — the node type denotes (intensionally) an operation constructor. Each node has links going upwards and downwards: a downward link points to an argument of the constructor, and an upward link means the node is an argument of some other node. Intensionality is

[1]The application contains 40 files, 8 packages, 154 classes and 872 macros / functions / methods, all this in about 13,000 lines of code.

provided by the fact that nodes are objects and links are pointers to nodes: one node can be a shared argument of more than one other node.

In the standard-iconic graphic environment, nodes are called *icons*[2]. These two terms are used equivalently from now on.

Node Features

- Node objects are instances of classes, each representing an editor domain operation. Nodes have argument lists that can be of three types:

  1. A fixed number of arguments, each one has a distinct symbolic name.
  2. An argument accepting a list of values (node links).
  3. No arguments — this case represents a leaf node.

  Nodes with list arguments are useful for many operations. These nodes can evaluate with no arguments connected, representing a future intention to fill this node. (Fixed argument nodes can usually be evaluated only when all arguments are full.)

- Argument links are pointers, and because nodes are *objects* with *changeable state-value variables*, the intensional behavior is automatic. For example, when the arguments of a node change, the flat-form evaluation result of all parent nodes will change. This also enables subexpression sharing.

  Usually, the arguments of a node will be links to other nodes, however, there is an option for having simple value arguments. This simplifies editing, for example, if there is a general 'stretch' musical operator: we don't want to force the user to manipulate number leaves whenever this node is used. The number can be a direct argument of the node (using the node edit dialog box).

- The node link mechanism includes a simple type-checker. Each node has an 'evaluation' function with a declared output type, and each argument of a fixed-argument-node, or the single one of a list-argument-node, is declared to accept some type. This applies to types of linked nodes as well as types of simple value arguments.

[2]The 'Icon' class is a subclass of the 'Node' class.

The type-checker allows an argument slot to accept only values of its declared types, or subtypes of it (in CLOS this includes subclasses). This is a basic approach, similar to specialized method argument lists in CLOS, but it is enough for considerably improving the hierarchy consistency.

- Almost all node-related functions are methods, allowing specialization on all node behavior aspects — from argument handling to type checking and evaluation.

  This allows a customizable node graphic user interface management — colors, fonts, events and specialized property sheets. A good example for this is the text icon which is quite different from standard nodes — it cannot be linked to any other nodes, and it looks different.

  Nodes interact with other nodes by means of message-passing. This is also using specialized methods — like "evaluate" messages flowing down the hierarchy and "forget-value" messages flowing up.

- Nodes use memoization for optimizing evaluation — nodes never get evaluated unless their value is actually needed, and this value is cached. This, however, cannot be called *lazy evaluation*: the current implementation evaluates during evaluation of some parent node evaluation, forcing it to get all of its sub nodes values recursively. This is an *applicative order* evaluation method.

  A minor modification to this evaluation scheme is to get the value of some sub node argument only when this value is actually used — enabling *normal-order evaluation*. This is effectively equivalent to the graph-reduction methods in [23]. This is implemented in BOOMS as an optional argument to the node class declaration. Usually, simple domains that are meant to be used by novice users are implemented in BOOMS, so such a feature is not important: it takes a sophisticated computer-theoretic user to catch BOOMS in an infinite loop using only abstractions...

- Structure abstraction is probably the most powerful feature of hierarchical editing. These are fully described in Section 5.3. Structure abstractions are extensively used in BOOMS; many supporting mechanisms are provided for a powerful, yet intuitive user interface on one hand, and simple, reusable program implementation on the other hand.

These abstractions are used a several ways:

- The main usage of structure abstractions is the capability of having user-defined nodes. This is the usage intended in the first place. The graphic user interface supports this by having abstractions named and placed on the node class palette for later usage.
- Another usage for structure abstractions is zooming out — the way users focus on some part of the hierarchy, not having too much extra visual distractions. This is supported by the graphic system by allowing unnamed abstractions to be created and immediately replace the original icons. Actually, named abstractions behave the same, but having unnamed abstractions is useful for zooming out.
- It is sometimes desirable to have a useful group of icons named so creation of more copies of this group is possible. Abstractions can be used for this — abstracting the icon group, then instantiate the new class in other places and exploding it (explosion is the opposite operation of abstraction). However, This is a misuse of the abstraction feature: there was no intention for abstracting on the first place, all that was needed is a name for this icon group and the possibility to have more copies of its structure. Neither minimization of the group to an icon nor naming this icon as a user-defined operation is needed. Another disadvantage of this method is the fact that only node groups with a specific limited form can be used for creating an abstraction, see Section 5.3.1).

  This usage is provided by the *named copies* feature. This is basically identical to the copy / paste facility, except that the copied set of icons can be given a string label for later pasting. No abstraction is created in this process, and the label appears in a special ("named-copies") section of the node class palette.

6.2 Document Types

The BOOMS framework provides support for more than one editing domain implementation. This is enabled by the document-type mechanism — there are several packages implemented, each defines a separate editing domain.

Document Type Features

- Each document-type package defines a whole domain. This includes an implementation of a set of domain dependent node classes that represent the intensional equivalence of editor operations.

  User defined classes are associated with their original document type. This means that abstractions and named copies are available only in documents of their original type.

- Window functions are methods that are specialized on document types. This is convenient for having specialized evaluation visualization, different node menus and node properties etc.

  Another possible use for this is having a different user-interface implementation — such that uses the double view approach described in Section 6.3.

- There are currently three built-in packages implemented in the BOOMS system:

  Arithmetic This sample package handles intensional representation of simple arithmetic expressions, it is described in page 23. The flat-forms handled in this domain are natural numbers.

  GCalc This package fully implements the GCalc domain described in Chapter 4 with abstractions from Section 5.2. Flat-form values are conceptual cube values. This is the only package that implements a dual-view editor (flat-form / hierarchical).

  Music This is an implementation of the original motivating domain — the music domain. The flat-form values of this domain are flat music pieces, and 'visualization' is done by audio playing the piece. As a music editor implementation, this package is extremely simple — only very basic constructors and functions are implemented. Many more features can be implemented, as seen in computer composing tools that exist today.

6.3 Multiple View Editing

BOOMS provides the needed basis for implementing a multiple view editor.

Multiple-view editing means having several editing windows, existing simultaneously for the same data. Different windows can represent data in different ways, and operate using different user-interfaces. This supports a selective view, and enables coexistence of a graphic, textual, and acoustic user interfaces. Multiple view editing is demanding, since it requires intimate interaction between a view and its underlying data — there is a need for thoughtful separation between knowledge levels.

Implementing multiple views is simplified by the "Document-View" approach. This was developed in the 1980's in Smalltalk as a way of organizing graphic user interface programs [25]. It is best implemented using object-oriented programming. The central idea is having two objects interacting one with the other:

Document This object represents the actual data. Operations on data should be implemented as methods of this object.

View The view object defines the graphic user interface to the document. The methods defined for view objects are graphic input/output methods — visualizing the data, and graphically perform operations on it.

The document object can be thought of as implementing the ontology, and a view object as a syntactic tool for accessing it.

The document-view approach has many organizational benefits, one of them is the simplicity of having multiple views representing the data object. Each view visualizes the data objects in some way, and changes in the data objects are reflected to all of its views. This allows having two different view windows opened side-by-side, and working in one window, observing the changes in the other.

This is used in BOOMS to integrate flat-form editing with hierarchical editing — BOOMS has an internal data structure which is the operational hierarchy, and a built-in hierarchy editor. The GCalc package uses the extendibility of BOOMS to incorporate a second view which is an implementation of the GCalc flat-form editor. Work is therefore possible in the hierarchical world of operations as well as in the GCalc native editor.

The Reflection of the history of flat-form editor operations in the BOOMS hierarchy allows for 'users education'. The natural editor for a beginner will be the flat-form editor, which supports intuitive work. As work advances, the user might explore from time to time the hierarchy created, up to the stage

he understands it. At that point, the user can start reorganizing hierarchical expressions to reflect his intentions (like sharing subexpressions or having the hierarchy based on logical subunits). Finally, when becoming an advanced user, one should use the flat-form editor only for brief sessions, creating the "building blocks" for more complex creations. The musical domain equivalent of this is using the keyboard for entering very short pieces and working with the musical hierarchy combining them into larger pieces.

6.4 Miscellaneous

There are many more features in BOOMS, many of them are dedicated to an implementation of a smooth user-interface (see Section 6.5) and program-interface (Section 6.6). Few of these features are described below.

- BOOMS uses an extensive graphic user-interface. This includes the usage of many Windows features like configurable graphic node attributes, right-button menus, palette, tool-bar and status-bar.

- There are utility node classes that are used in all document types — like identity nodes (for intensional abstractions), text label nodes (for graph decorations) and Lisp-value leaves.

- Copy / paste operations are implemented. These handle the subgraph defined by the selected set of icons, including its internal links and excluding its external links.

- Saving and loading operations are also implemented. A full document hierarchy is saved with its used named abstractions (a document containing several named abstractions can be used as a user-defined functions library), and the window attributes (location, size etc.).

- Finally, BOOMS was developed for the final goal of implementing music editing, therefore it has built-in MIDI support. A MIDI driver (in the form of a DLL) written in C and a Lisp interface to this driver are implemented.

6.5 User Interface Considerations

BOOMS was designed around the main concept of being a general editor that allows hierarchical manipulation of creative domains. This emphasize usage by creative people, which are usually novice computer users. There are, of course, creative people that are used to do their work on a computer, but the majority needs intuitive user-friendly editors.

[26] is a good source for user-interface discussions. A major subject of this book is "direct manipulation" which means that a good user-interface should provide direct manipulation of the work domain. In BOOMS, we manipulate both the flat-form domain and the structure representing the user intentions — by having the native domain editor merged into an environment that supports structure (historical) editing.

Another aspect is having an "Extreme User-Friendliness" philosophy — as demonstrated by this quotation:

> When an interactive system is well designed, the interface almost disappears, enabling users to concentrate on their work, exploration, or pleasure. Creating an environment in which tasks are carried out almost effortlessly and users are "in the flow" requires a great deal of hard work from the designer. [26, page 9]

6.5.1 Expectations Detection

A reliable user-interface is crucial for an editor to be accepted by users. A good example is the demand for bug-free commercial software: when working in the commercial software industry, it is often the case that an application is presented for sale before all bugs are wiped out. When a user encounters such a bug he will usually regard this software as unreliable, and not use it even after the bug he encountered is fixed. This will make most users stay with the application they are used to. Creative users that use computers for the first time will probably abandon the whole idea of computer editing.

An important aspect of having a reliable user-interface is expectations detection. This means that if something is expected by the user, then the system fulfills this expectation. One of the implications of this is having appropriate defaults whenever needed — the system will provide a default behavior when it is expected. An example of this is the operation of linking nodes: usually, a line should be stretched from one node's socket to another,

but the system guesses the destination socket when the line is dropped on an icon body (this is using the specified argument types declarations for nodes). However, defaults are not used excessively. For example, a possible user-interface implementation can use automatic graph layout for icon placement. BOOMS takes the opposite approach of having the user use his own preferred layout — icons are placed by the user. This seems like a better interface for users that use different placement schemes as another expressive dimension.

Another aspect of expectation detection is having the system merge into the graphic environment. Users of a graphic windowing environment tend to get used to standards in this environment, and take any feature that is not standard as annoying. In the BOOMS case, the graphic environment is Microsoft Windows, so many features are implemented. Some of them are: pull-down menus, pop-up context sensitive menus, tool-bar, status-bar and standard accelerator keys. These are described in the next section.

6.5.2 GUI Usage

Graphic User Interfaces such as Windows have a critical role when making a software package available for novice users. This means that when designing such applications, the GUI should be used as much as possible — this is also the case of the BOOMS system.

BOOMS uses Microsoft Windows as the user interface environment. This environment is designed to provide users with convenient and standard application environment — there are many conventions that Windows applications follows. The BOOMS system is no exception for this — the implementation uses as many standard features as possible. A detailed list is given:

- Menus are the basic way for a user to send command to a Windows applications. There are menus of two kinds — pull-down menus and pop-up menus. Pull-down menus are the standard menus that are attached to the top of the window, these contain all BOOMS commands. BOOMS also uses context-sensitive pop-up menus, described in the next item.

- When the right mouse button is clicked, a context-sensitive menu pops. This is a standard feature in Windows that is implemented in BOOMS. There are different menus for the BOOMS window, document windows and nodes.

Node menus are chosen using a specializable method, so these menus contain node specific operations.

- A tool-bar is also used in BOOMS — this is a small bar on the top part of the BOOMS window that displays some buttons and controls that provide convenient shortcut to frequent commands.

- The status-bar is a display area at the bottom of the BOOMS window which is capable of displaying relevant messages to the user. This serves several purposes:

  - Help messages are presented for menu items and tool-bar buttons.
  - Provide visual feedback for long operations like load / save.
  - Various informative messages are presented, such as error messages and a description of the current node / socket the mouse pointer is located at.

- Accelerator keys are used as yet another way of shortcuts to common commands. These include several key combinations that are used in all window applications (Control-S, Control-O etc.) and other keys that are specific to BOOMS.

- A common feature among modern applications is property sheets. BOOMS enables node editing through edit dialogs that resemble property sheets.

  The edit dialog mechanism allows definition of edit panes that will operate on node properties (these might be specific to the package used). Then, a node method can choose the pages available for a node. Two standard pages are implemented — the node edit page (editing node arguments using a dialog) and the colors edit page.

- The management of nodes and sockets means that the user should know what socket or icon he is manipulating. This can be unclear in some situations, where there are a lot of icons crowded in a small place. BOOMS makes this easy by indicating the current icon and socket in the status-bar as well as a graphic indication — the node which is under the mouse pointer (if any) is highlighted, as well as the socket pointed. This is not confused with the highlighted icons that are the selected icons for the next icon-group operation.

- Floating palette windows are very useful in applications that provide selection of a tool from several available, like selecting the graphic tool to use in a drawing software. In BOOMS, such a selection is made of the node class that is used for creating instances. This is made from a floating palette that contains a tree of selectable node class names. This tree is not based on node classes inheritance trees, but on a hierarchy defined only for this user interface.

  This palette aid the user in three ways:

  1. The user can choose a node class to create instances of.
  2. The palette provides a visual indication of the current selected class.
  3. An optional, sometimes more convenient use of the classes palette is for both selecting a class and instantiating it — by clicking on a class name, and dragging it to the document.

- The Windows standard "Document-View" approach is used by BOOMS (this is described in Section 6.3). This has an effect mainly on the program structure, the effect on user interface is that it is generally easier to implement standard Windows behavior, like multiple views. Using this approach in BOOMS, adding more Windows features like having several windows for the same document can be added.

Graphical Ethics / Esthetics

Another aspect of GUI usage is that it should be "well-behaved". This is demonstrated by several points:

- 'Affordance' — this means that the way graphic objects should be used should be obvious by their visualization. For example, the appearance of a button is enough for saying "Push me!" to the user. Example for this in BOOMS is the visualization of node icons that makes linking them a natural operation.

- A different point is the graphical ethics — everything should be convenient enough for a novice user. This generally means that the program should be complete in the sense that all details were worked on. This is demonstrated by having a movement threshold for triggering a drag

operation (a system that moves objects on every random click is highly annoying), or by having the load operation restore the window position, size and state that were actual when it was saved.

6.5.3 Visualization Considerations

We discuss now the visualization that is used in BOOMS. Good visualization relies on a good system design — the system should be implemented in a way that is faithful to the ontology domain. There is a tricky point that should be fully understood here: when using flat-form editing, the ontology is the domain objects, a hierarchical structure defines some domain objects *implicitly*. However, when using the structure editor for manipulation of this hierarchy, the ontology is the historical operation structures — a hierarchical structure is an *explicit* object in this domain.

Direct Manipulation

An system that uses "Direct Manipulation"[3] is a system that enables working with a direct [visual] representation of its domain values. This is especially important when using editing systems. A good example is the full-page display editors. Such visual editors improved the world of text editing when they replaced the line-oriented text editors that were once common. The difference is that using visual editors, one can immediately see the result of his work — the screen reflects the final text file. Using a line editor a print command should be issued in order to see the result.

Most editors cannot be as direct as visual text editors. The representation used by a visual text editor is truly direct — the screen shows exactly what the printer will print. Other editing domains settle for some visualization which is as close as possible to the domain. Using direct manipulation makes an improved system — the system is more intuitive thus easy to learn and master.

The implementation of direct manipulation in BOOMS is double-faced. We have a flat-form editor, which naturally uses a visual representation of the domain object. The hierarchical editor should also implement direct editing of hierarchies. Using a visual DAG representation made of iconic nodes is ideal for this purpose — this is the natural way for users to think of

[3] Direct Manipulation is described in length at [26, Chapter 5].

hierarchies. Other approaches like windows-inside-windows, frame diagrams and HyGraphs tend to be more complex for users (and programmers). Some of these other approaches can be found at [27].

Conceptualization

BOOMS is an editing environment that should support the concept of hierarchical editing. Programming such an environment is simplified when conceptualize in the "right" way. Here the term 'conceptualization' refers to the way the program represents its real-world / user-interface. The BOOMS project uses CLOS classes for this purpose (see Section ADVANCED-PROGRAMMING-LANGUAGE). The program is well formed into separate layers, each represented by several classes. Two important layers are the internal data structure layer and its visualization — this is built as a stand alone class that is used to add the GUI capabilities to the internal object (using inheritance). This results in a stable user interface, for example — user defined functions behave like any other node class.

6.6 Program Interface Considerations

One of the main goals of the BOOMS project is to serve as a basic application that can be easily expanded by other programmers, for implementing more domain editing packages. The system should therefore be "Programmer-Friendly" — the application should be:

smooth The program should be well-structured, it should be layered, contain independent layers using a well-defined interface and functions should be short and understandable. Bad programming makes frustrated programmers that will not use it as a basis for their extensions.

robust A system that is meant to be extended should stand to major shocks. Bad systems usually demand total rewrites when some basic feature is to be added or modified. A good example for such a property is Emacs Lisp programs — the hooks facility is commonly used to provide a code that can be customized without rewriting it. Object-Oriented programming is another technique that can be used for code customization, especially when implemented in a functional language (manipulating functions as first-class objects). BOOMS uses CLOS which is very rich

in this area (for example, `:before`, `:after` and `:around` methods for customizing methods).

powerful BOOMS should provide enough power, so people will actually *want* to use it as an editor. There is no point having this whole project if the final editor is so poor that people don't like using it. A good example is one of the first versions of BOOMS that was so slow that it could not be used for any purpose other than demonstrating a vague general idea.

extendible Finally, the system should lend itself to external programmers wishing to extend it. In general, every part of the program was designed with a thought of usage by other programmers. General mechanisms were implemented all over, a thing that made the program huge, but very powerful: many features can be used as a basis for many other programs, not only structure-editing applications.

These features should make the application have a longer life cycle. The main advantage is having a well-designed program. This can be compared with the Java programming language, that is a well designed language that is intended to serve as a common extension language: more and more institutes teach Java today as a first language, instead of the outdated Basic language.

6.6.1 Advanced Programming Language

The selection of a programming language is important when programming any application, but it can be a major issue for an application that is to be extended. Imagine a programmer reading the BOOMS sources for implementing some structured editor, now imagine the sources contain assembly code...

An advanced high-level language is needed for fulfilling the properties we want BOOMS to have. The language chosen for implementing BOOMS is Common Lisp, the Lisp version is Franz Common Lisp for Windows. Lisp was chosen due to many reasons:

- Common Lisp is one of the most advanced programming languages that can be used to write GUI applications. It features CLOS, an object oriented extension that is the first ANSI standard for an object-oriented language. The language contains many useful features, the official specification is extremely long, see [28].

In the same time, Lisp can be used for low-level optimizations — examples are macro expansion and system calls. The code is compiled using built-in incremental compilation facility.

- Lisp supports *rapid application development* by allowing the programmer to focus on global, top level programming rather than going down to local, system details (using C++ one always bumps to system level issues). This is compensated by having a slower application, but modern computers, and careful programming solve this problem.
- Using CLOS is a big advantage. The implementation is fully object oriented: classes ad methods are constantly used, so every aspect of BOOMS can be customized. This enables many of the features we require.
- The code is kept in mostly small self-explained functions. This is aimed at allowing external programmers to answer most of the questions that arise by simply reading source files. Building large applications from small functions is conventional in Lisp.
- Functional programming for extension language — functions as first class objects. Useful for a future music library — implementing a an algorithmic nodes, containing Lisp code.

6.6.2 Technicalities

Lisp applications usually suffer a certain amount of performance deficiency. This is shown both in two aspects:

Runtime Lisp programs tend to be slower than C-dialect programs. Some of the reasons for this are:

- Runtime type-checking is very slow compared to compile-time checking.
- A garbage collector makes life simpler, and program hang-ups — this is highly annoying during interactive operations.
- Using powerful tools for performing a simple tasks (*e.g.*, using `equalp` instead of `eq`).
- Using many small functions with no in-lining is slow.

Appearance Lisp GUI programs have a general tendency to look less polished than C++ programs. This is due to the language focus on conceptual matters rather than concrete system oriented programming — Lisp programmers are generally less aware of low level / operating system issues (this affects runtime performance as well).

BOOMS, being an interactive GUI application, cannot afford bad response time, or a amateur user interface, so these problems are specifically handled. (The many advantages of Lisp make the solution of using C++ less attractive.)

Efficiency

As said above, Lisp code can be heavily optimized. Here we encounter a conflict: optimized Lisp code makes it lose its clarity which is another feature we want to have. The "golden path" that is used in BOOMS, is optimizing the code without using declarations. Two important issues are heavily attacked:

1. The redrawing mechanism is crucial for interactiveness. An early version of BOOMS that was programmed on MCL (Macintosh Common Lisp) was a failure as an interactive system due to very bad redraw time. The manipulation more than of 10 icons took about one second redraw time per icon — a fact that made it useless as an interactive editor.

   A lot of effort was therefore invested in the current implementation. This involves geometrical functions, and Windows specific functions — delaying redraw events, and drawing on a part of the screen. BOOMS can can handle now hundreds of icons with acceptable redraw time.

2. GC is a general problem of functional languages, and Lisp is no exception. Avoiding unnecessary GCs was mainly achieved by using preallocated objects and destructively modifying them. This is done by heavily using the "`#.`" Lisp construct (see [28, Section 22.1.3] for a formal definition). A code example demonstrates this best[4]:

   ```
   (defun NNBOX-TOP-LEFT (box)
     (let ((pos #.(make-position 0 0)))
   ```

[4]These examples use two Allegro Lisp data types: `position` and `box`. Their usage should be obvious.

```
    (nmake-position pos
                    (box-left box) (box-top box))))

(defmethod ICON-SET-POS ((icon icon) (pos position))
  (let ((offset #.(make-position 0 0)))
    (ncopy-position offset pos)
    (nposition- offset (icon-center-pos icon))
    (dolist (box (icon-box-list icon))
      (nbox-move box offset)))
  (set-modified (icon-doc icon)))
```

The first function returns the top-left corner of a given `box` value which is a `position` object, but each invocation of this function returns the *same* [modified] object. This can be dangerous — for example, one expression cannot use two calls to this function. The second function hold a single preallocated `position` value, and modify it on every call.

This is the general approach used, especially in interactive functions like the mouse-move event handler and the drag-and-drop mechanism.

Graphics

Much work was invested in making a smooth user interface. This means that although the main focus is on the conceptual level, the program is never "too clean to get into system details". This help achieving the global goal of having users believe in the system (Windows users will not take any non-standard application as reliable).

This includes working around bugs found in the Franz Lisp interface code, and code fragments that handle things that does not look important (like event handlers and menu functions). A good example is the color-box button widget: there was a need for some dialog widget used for color selection. Using a standard button does not allow visual indication of the current color, making bad interface, so a whole new Lisp widget was designed and implemented.

Chapter 7

Conclusion

Hierarchical editing and structure abstractions are the main features of the BOOMS application framework. Hierarchical editing enables direct manipulation of structured *objects* with an operational meaning. Hierarchical abstraction is a user-interactive way for creating new functions. These can be compared with keyboard-macros that are used in text editors, since both are created by naive users during an editing session. Both create objects that are recordings of edit operations for later reusage. Using BOOMS abstractions is more powerful since it allows recording of operation *patterns* rather than *flat sequences*. BOOMS supports explicit management of operation patterns, allowing editing of abstractions.

The BOOMS project shows that the multiple views approach can be used to design robust user editing environments. The interaction between different modes (flat-form and hierarchical) confronts different expressions of conceptual operations. In BOOMS we saw that with respect to the abstraction and application operations: they were treated *differently* in the flat-form and in the hierarchical editor. The interaction resulted in interesting fine distinctions.

The extension of the flat-form editing with the BOOMS system was relatively smooth, due to the clean separation between levels of knowledge in BOOMS. In particular, the strict separation between visualization and internal structuring proved essential. The demanding music application played a crucial role in the development of BOOMS, since it was not possible to give-up hierarchy, intuitive visualization, or abstraction.

Of the three package implementations in BOOMS, the music package is the only one that can have concrete usage. Indeed this package deserves

extensive enhancements.

7.1 Future Music editing Development

BOOMS is far from being a musical editor that composers will choose as a working tool. The music package that is currently implemented is only a demonstration — this should serve as a basis for a fully functional music editing package.

Here is a list of features that can be added for an implementation of such a package:

Rich Musical Libraries A rich musical library is found in all music editing tools used today. Such a library should include many useful functions and data types. It should allow more than simple MIDI music and more than a single visual representation of music pieces. A good example is [13]. A CLOS music knowledge base for fundamental concepts in tonal harmony, based on a concise arithmetic for notes and intervals in the tonal system, is also in preparation.

Musical Flat-Form Editor As discussed in Chapter 4, a flat form editor is important for novice users. Such an editor should be implemented as part of a functional music editor — it can be based on any existing conventional editor.

Multiple Views for Music Editing Views other than the hierarchical and the flat-form views can be useful to implement. Each is convenient for different type of editing. The view types can contain:

1. A BOOMS hierarchy view (the built-in editor).
2. Flat form editing, based on a score view.
3. Procedural representation — Lisp or Scheme code.
4. DMIX-like bar graphs.
5. MIDI event editor.
6. Music lambda calculus [19].

Streams Streams are very useful data types for computer-music compositions. A good example for good stream usage can be found in the Common Music editor [20] and also in the editor implemented at Grame[14].

Algorithmic Nodes Nodes with a function attached, these can be useful for creating algorithmic music — allowing usage of loops, random notes conditionals and more. Scheme will be perfect language for this.

Real-Time Having real-time editing — use the system as a performance tool.

Attribute Management The idea, presented in [29], is decorating musical nodes with various attributes to enhance expressiveness. These attributes propagate through the hierarchical DAG in several ways:

- Timbre attributes propagate up.
- Author and performance directives propagate down.
- Velocity is accumulated going upward.
- Tempo-setting attributes have a side-effect whenever reached during playing of music pieces.

Appendix A

BOOMS Users Manual

A.1 Platform

BOOMS was written on Franz Allegro Common Lisp for Windows. It is *not* a stand-alone application, so Lisp should be purchased and installed. The program uses a lot of Windows functions so porting it to other operating system is not possible. The project was written and tested under Microsoft Windows 95, no other Windows versions were tested, but generally it should work on any 32 bit Windows.

The recommended environment includes:

- A PC with 8Mb RAM or more and fast CPU are preferred features.
- Windows 95.
- Allegro Common Lisp for Windows — version 3.0 and above.
- Video resolution of 1024×768.
- BOOMS does not require much disk-space (Allegro Lisp does that).

A.2 Getting BOOMS

FTPing The whole system can be found at

`ftp.cs.bgu.ac.il:/pub/people/eli/booms.exe`.

This is an ARJ self-extractor file. Here is a sample session for getting BOOMS (user input in bold face):

```
mycomputer> ftp ftp.cs.bgu.ac.il
Connected to lace.cs.bgu.ac.il.
220 lace FTP server (Version wu-2.4(7) Wed May 1 20:16:54 GMT+0200 1996)
ready.
Name (ftp.cs.bgu.ac.il:eli):  anonymous
331 Guest login ok, send your complete e-mail address as password.
Password:  ...
.  .  .
230 Guest login ok, access restrictions apply.
ftp> cd pub/people/eli
250 CWD command successful.
ftp> bin
200 Type set to I.
ftp> get booms.exe
200 PORT command successful.
150 Opening BINARY mode data connection for booms.exe (563818 bytes).
226 Transfer complete.
local:  booms.exe remote:  booms.exe
563818 bytes received in 1.4 seconds (3.8e+02 Kbytes/s)
ftp> quit
221 Goodbye.
mycomputer>
```

If any problems occur, send mail to `eli@cs.bgu.ac.il`

Extracting The system should be extracted to some (preferably new) directory, this is done by running the self-extractor. For example:

```
C:\>  cd allegro
C:\ALLEGRO>  md booms
C:\ALLEGRO>  cd booms
C:\ALLEGRO\BOOMS>  copy a:booms.exe
        1 file(s) copied
C:\ALLEGRO\BOOMS>  booms.exe
. . .
```

A.3 BOOMS Loading / Compiling / Running

First thing you should make sure that the valid file types variable, `*load-pathname-types*` is bound to `'(#p".fsl" #p".lsp")` (this is useful anyway), and that `*default-pathname-defaults*` is bound to the directory containing BOOMS. Changing the Lisp working directory is another thing you would probably want to do. This can be done by having these three lines:

```
(setf *default-pathname-defaults*
      #P"C:\\ALLEGRO\\BOOMS\\")
(setf *load-pathname-types* (list #p".fsl" #p".lsp"))
(set-current-directory "C:\\ALLEGRO\\BOOMS")
```

added in `C:\ALLEGRO\STARTUP.LSP` file (assuming this is the Lisp installation directory).

Once you've got this done, loading, compiling, and running BOOMS is simple, each operation is performed by loading a file indicated in this table:

Loading	`_LOAD.LSP`
Compiling	`_COMPILE.LSP`
Running	`_RUN.LSP`

If the system is already loaded — the function '`booms`' can be also used for running it.

Note that Allegro Lisp for Windows automatically compiles loaded functions, so compilation has affects only load-time, not run-time.

A.4 Using BOOMS

This section provides a brief description of using the system. This is based on the assumption that the reader is already familiar with the hierarchical editing ideas as described in this work. Specifically, these subjects should be clear:

- DAG representing an editing session operational hierarchy.
- Management of such a DAG.
- Creating and using a structure abstraction.

Anyway, this is just a brief description of BOOMS operations available to the user, it is not a replacement for actual experimenting.

A.4.1 General Hierarchy Editing

The BOOMS hierarchy editor manages operational hierarchies based on an editor domain. Each document in BOOMS (a file) contains an operational hierarchy DAG consisting of nodes.

Nodes

A node is represented by a small rectangular icon with sockets that can be used as links to other nodes. Nodes are created by double clicking a clear spot on a document window, or by dragging a node-class name from the class palette (see below).

Manipulating nodes is as simple as manipulating icons in any standard Windows applications. Selection is done by clicking nodes or by selecting a surrounding rectangle. If the shift key is held down, then the selected icons are toggled rather than selected.

When the mouse pointer is over an icon, indicative text is printed on the status-bar, such as the node type, arguments and value. The icon is also highlighted to make sure that close nodes are not confused.

Double-clicking a node causes it to evaluate, this is visualized in some way that depends on the document type. Right clicking pops a node menu, that often contains operations specific for this node. Using this menu, or clicking the node with the Control key down pops up a node edit dialog that is used to modify the node properties.

Sockets

Nodes have sockets near the top edge and near the bottom edge of the icon. The top socket is the output-value socket and the sockets on the bottom are input argument sockets. There are three types of argument sockets for nodes:

1. A node can have a fixed number of arguments. An unconnected socket remains visible, and the node cannot there are any.

2. A node can have a list of arguments. Such a node can always be evaluated. When a link is deleted, its socket is deleted. The number of

visible sockets is always one more than the actual arguments connected: the extra socket is used for connecting new node arguments.

3. Finally, a "leaf" node is a node with no sockets at all — it is bound to appear at the bottom of the DAG.

Sockets are typed: the top socket determines the node evaluation value type, and the bottom argument sockets have types that determine what output sockets can be linked to them.

Manipulating argument links is simple: connecting two nodes is performed by dragging a line from one node input (output) socket to the other node output (input) socket. If the link is dragged to the socket body, then some appropriate socket is selected by a default method. When the mouse pointer is over a socket, the socket is highlighted indicating a link drag operations can begin, and the descriptive node description line on the status bar is modified to show extra information on this socket. Deleting a link is done by clicking its top socket to a clear document part (the bottom socket cannot be used for this as it is an output socket and a new link will be created instead).

Finally, an alternative way of modifying node arguments is by using the node edit dialog box. This is the only way to specify constant [Lisp] values as being node arguments.

A.4.2 Document Types

A document type package is an implementation of some specific editor domain, meant for hierarchy editing implementation in BOOMS. Each document type package defines a whole environment. The main definitions are for specific node types, with some additional customized code like specific menus and edit panes. Three document types are defined:

Music

This implements a simple hierarchical music editor. It is based on the Music Structures approach. This package is intended to demonstrate the possibility of a music editor implementation, much work should be invested to make this an editor that can attract composers.

The node classes defined in this package are simple music concatenations, and some basic functions. Flat values are flat music pieces — double-clicking an icon visualize this by playing the music piece on a MIDI channel.

GCalc

The GCalc package implements the cube world defined at Chapter 4, based on [19]. This package is the only package in BOOMS implementing a dual editor: there are always two views connected to the same document, one of them is a "flat-form editor", and the other is a hierarchical editor.

Node classes are cubes constructors, flat-form abstractions and application. Double-click visualization is done by displaying a window with the cube drawing.

Arithmetic

This package implements the arithmetic example from Section 4. It is a simple example of a package implementation in BOOMS, that can be used as a basic template for implementing more packages.

Node classes are arithmetic expression constructors. Evaluation of these nodes yields an integer number, which is displayed in the status-bar upon double-click evaluation.

Shared Icon Classes

There are several shared node classes that are available in all document types:

Identity Nodes These are icons that output the same value connected to their input. They are useful for structure abstraction — when it is desired to have the opposite behavior of the "Same-Node-Same-Arg" option:

When an icon list that is about to become an abstraction have one argument connecting in more than one place, the Same-Node-Same-Arg option will determine the result. If this option is on, then a single-argument abstraction is created, otherwise each occurrence will have a corresponding argument socket.

An example will make this clear: if the expression $2 + 2$, having the same 2 object connected twice to a binary plus node, is abstracted on the single 2 object then the result will be:

- $X+Y$ if Same-Node-Same-Arg is off, and the two argument sockets X and Y are connected to the 2 object.
- $X + X$ if Same-Node-Same-Arg is on, and the single X argument is connected to the 2 object.

The intelligent reader can see that an identity node can be used in both cases to have the opposite behavior.

Lisp Value Nodes Constant nodes contains a single Lisp value that is used as the value of the node. They can be used only by a user that is aware of the underlying Lisp implementation and data types used.

Flat-Form Abstractions / Application These node types implement the flat-form abstraction and application operations. They are available only when the document-type package defines the needed search and replace operations.

Text Nodes These are non-conventional nodes that cannot be part of the operational hierarchy. They are used as text labels for decorating the hierarchy with short, descriptive messages.

A.4.3 The Class Palette

The class palette is a small floating window that is placed at right side of the screen. The palette shows all available node classes, hierarchically ordered in a collapsible tree. The hierarchy is not determined by internal aspects such as class inheritance, it is only for user-interface convenience.

The palette is used in three ways:

1. It gives a visual indication of the current node class selected.
2. Clicking on a class name makes it the current node class (double-clicking will create instance of this class).
3. The two operations of selecting a class and then creating an instance, can be compacted into a single click-a-class and drop-an-instance operation.

The classes menu is a equivalent to the classes palette and can be used instead.

A.4.4 Abstractions

Abstractions are created by selecting some icons and performing an abstraction operation. This icon groups must stand in these conditions:

- The icon list compose a connected subgraph.
- This subgraph have a single root node (for determining the abstraction type).
- All external links (going from nodes in the icon-list to other nodes) are either from the root node upward, or from leaf nodes downward (leaf nodes in the selected subgraph).

The abstraction created is an icon with the same output type as the root node of the abstracted icon group. This node contains argument links that correspond to the external nodes that were connected as arguments to the group icons. If the Same-Node-Same-Arg option is set, then each argument object will have a matching abstraction input socket. If it is not set, then each link will have an argument socket — if the same object has two links to the group icons, two argument sockets will be created.

The created abstraction icon immediately replace the original group, and its input and output sockets are linked correctly so the DAG has basically the same values.

If the abstraction is named, its name will be added to the node class palette, under the user-defined label. In this case, more instances of this abstraction can be created. Named abstractions are labeled by their names, so they appear as a user defined function (abstraction names should indicate their operation).

Abstractions can be exploded by the user. This means the original node subgraph will replace the abstraction (with arguments correctly linked). This can be done for an unnamed abstraction as well as on a named one.

This facility might be misused as a way for naming a subgraph for later using it (as a subgraph, not as the function it performs). This is better done with the named-copy mechanism. This allows selecting a group of icons and giving it a name that will appear in the named-copies section in the class palette. This name can be later dragged to a document for creating another copy of the original group. This is better for the purpose of naming a subgraph and having more copies of it because:

- Named copies can be created from any group of icons.
- No need to abstract first and then explode.
- The operation is simple and quick — no tests are made.

A.4.5 Menu Commands

File Menu

File management operations:

New Create a new document. This will be done only after inquiring the user for the new document type — selecting from the three packages currently implemented.

Open Open a saved document. BOOMS will automatically identify the document type and auto-load all named abstractions to the classes palette.

Close Close the active document. Offer to save the document if it was modified.

Close All Close all documents. Offer to save modified documents.

Save Save the current document. The saved information will include:

1. Document type.
2. Current DAG graph.
3. Named abstractions used
4. Window location, size and state.
5. Option settings.

If the document is saved for the first time, then the user is prompted for a new name.

Save As Save the document under a new name.

Save All Save all documents.

Exit Exit BOOMS (offer to save modified documents). This will only exit the current BOOMS session, not the Lisp application.

Edit Menu

When reading the documentation of clipboard operations, the reader should note that BOOMS holds a separate clipboard for each document type.
Icon editing operations:

Cut Cut the selected icons to the BOOMS clipboard (for the current type). The icons are copied with their internal links (links between members of the selected icon list); external links are deleted.

Copy Copy the selected icons to the BOOMS clipboard (for the current type).

Paste Paste the icons in the BOOMS clipboard (for the current type).

Delete Delete the selected icons.

Edit icon Edit the current icon using the applicable edit pane list (see below for description of icon edit panes).

Grab style Grab (store) the current icon style. The icon style includes its colors (background, foreground and highlight) and its font. The font is effective only for text label nodes.

Apply Style Apply the current grabbed style to the selected icons. This means that the selected node[s] will have the grabbed style. The font style only applies only to text messages (only if the current grabbed style was taken from a text node).

Select All Select all icons in this document.

Clear All Delete all icons in this document. Warn the user first!

Abstract Menu

Abstractions operations:

Create Icon Iconify the selected icons to a single abstraction. This creates an unnamed abstraction — the selected icons are abstracted to an abstraction icon that replace them. No class name is added in the icon class palette.

Explode Explode the selected abstraction[s] to the original components. This applies both to named and to unnamed abstractions. The original nodes will replace the abstraction nodes, scattered around its location.

Name Name the selected abstraction — add it in the icon-class palette and menu. If multiple icons are selected, an abstraction is created first and then it is named.

Unname Unname the selected abstraction[s] — the selected named abstraction icons are turned into unnamed abstractions.

Named Copy Create a named copy — copy the selected icons to a special named register that appears on the icon-class palette. This can be later used for pasting instances of this selection. The conventional "copy" operation is identical to this operation when the name is "clipboard".

Remove Current Remove the selected user-defined abstractions or user-named-copy from the palette / menu. Removing a named-copy has no effect other than removing the selected entry, but removing a named-abstraction will unname all of its instances in all documents (this requires user-confirmation).

Remove Unused Remove unused abstraction entries from the palette / menu for this type. This will scan all named abstractions currently available for the current document type, and remove the entries for the unused abstractions. An abstraction is not used only if no document (of the current type) contains an instance of it.

Remove Copies Remove all named copies from the palette / menu for this type. Note that named-copies cannot be used in a document.

Classes Menu

This menu holds all icon-classes. It is equivalent to the icon-class palette. Only the sub-menu corresponding to the current document type is available.

Windows Menu

This menu contains two sections, one for toggling panes (showing / hiding them), and one for selecting a document window:

Toggle Palette Toggle the icon classes palette.

Toggle Status-bar Toggle the BOOMS Status-bar.

Toggle Toolbar Toggle the BOOMS Toolbar.

Document Window Selections These entries, one for each active document, select one them.

Options Menu

Various BOOMS options — these are general operations, and option setting for the current document:

Same-Node-Same-Arg This determine BOOMS behavior when abstraction to be created has an argument appearing in several places. If this is turned on, then each appearance will create another argument slot, and if it is off, then each *single* node argument will produce a *single* slot.

Dialog create Toggle creating an icons with a dialog — whenever an icon is created, an edit dialog box immediately pops up for editing the new icon. Pressing the "Cancel" button will undo the icon creation.

Grid Toggle grid placement for icons — if this is on, then icons can only be placed on grid locations, making it easy to organize icons "neatly".

Refresh Refresh the screen — redraw the whole screen in case of mess due to bugs (this should not happen).

Stop Music Stop playing MIDI music — clear the current MIDI music buffer.

Garbage Collect Run a Lisp garbage-collection.

Help Menus

General information menu, there is no "real help" here.

Welcome Hello from BOOMS...

About BOOMS This pops up a dialog box showing the contents of `BOOMS.TXT`, containing the BOOMS abstract.

Context Sensitive Menus

Context-sensitive menus are activated on a mouse right-click operation. There are three context-menu types:

1. Right-clicking the application window (the pane available between document windows) pops up a menu with these options:
   - Create a new document.
   - Open a document.
   - Close all documents.
   - Save all opened documents.
   - The whole Windows menu is also available.
2. Right-clicking a document area (not inside an icon) pops up a menu with these options:
   - Save this document.
   - Close this document.
   - The whole Classes menu is also available.
3. Right-clicking an icon pops its own specialized menu, this contain:
   - Evaluate, which is equivalent to double-clicking the icon.
   - Edit, which pops up the edit-icon dialog.
   - More options are available, according to the specific node type. These options are part of the document-type package.

A.4.6 Keyboard / Toolbar Shortcuts

Accelerator keys and toolbar buttons are a standard GUI feature. The Windows operating system has a lot of conventional accelerator key bindings, many of them are used in BOOMS.

A list of all accelerator keys is given below. "C-" means holding the Control key, "S-" means holding he Shift key and "A-" is the Alt key.

Key	Operation	Key	Operation
C-**N**	New Document	C-**O**	Open Document
C-**W** C-**F4**	Close Document	C-**S**	Save Document
A-**X**	Exit BOOMS	C-**X** S-**Del**	Cut
C-**C** C-**Ins**	Copy	C-**V** S-**Ins**	Paste
Del C-**Del**	Delete	A-**Enter**	Edit Node
C-**G**	Grab Style	C-**P**	Apply Style
C-**A**	Select All	C-**I**	Create Abs.
C-**E**	Explode Abs.	C-**M**	Name Abs.
C-S-**C**	Named Copy	**F10**	Palette Togg.
F11	Status-bar Togg.	**F12**	Toolbar Togg.
S-C-**D**	Dialog Create Togg.	S-C-**G**	Grid Togg.
C-**R**	Refresh Screen	M-**F4**	Close Lisp
Arrows	Move Icons	S-**Arrows**	Fast Icons
C-**Arrows**	Scroll Window	C-S-**Arrows**	Fast Scroll
C-**Tab**	Next Document		

The BOOMS toolbar provides many useful shortcuts — just point the mouse to a button and a message describing its operation will appear near it, and at the status-bar. The is one operation that is available only from the toolbar, volume setting (the right track-bar).

A.4.7 Node Edit Panes

A useful feature that is standard in Windows is attribute panes. These exist for many objects and can be popped by choosing properties from a menu (usually the right-button menu associated with the object) or by pressing "Alt+Enter". A property dialog box is mostly made using a tab control to switch different pages, each is used for some different attribute changing.

In BOOMS, the node edit dialog contain property edit panes. This feature is fully customizable by package implementations — each node can have its own set of property pages.

Here is the list of current edit pages and their usage:

`node` This is the most important edit page. It allows editing of node arguments — linking node arguments, or disconnecting them. This also provides the only possible way for filling an argument socket with a Lisp value.

This page is available to all nodes (not to leaves).

`leaf` This page is available to all leaf nodes. It controls the leaf value.

`colors` This page is available to all nodes. It is used to determine the colors used for nodes. The colors are the foreground and background colors used to draw icons with, and the selection highlight background color.

`text` This page is only for text message nodes. It controls the text message displayed, its font and its color.

`notated` This page is for editing notated triplets naming pitch or interval values in the music package.

`sound` This page is for editing a sound node in the music package.

`factor` This page is for editing music package factor values (real number between 0 and 1).

`g-color` This is used in the GCalc package by primitive cubes — selecting the cube color. Note that it is available to the general cube, constant-color cubes cannot change their colors.

`arith-name` This is a general page for all nodes classes in the arithmetic package. These nodes have a "name" that can be modified using this page. It is not very useful, but it is for demonstration purpose.

`operator` The arithmetic package defines general operator nodes, that can have one of several Lisp functions available. This page is used to change this operator.

Appendix B

BOOMS Programmers Reference

These appendix is not complete yet, currently there is only the files documentations.

B.1 Document Type and Icon Class Additions

As written in this work many times, BOOMS is designed to make additions of editing domain packages very easy. The arithmetic package implementation file, `ARITH.LSP`, is provided mainly as a template for learning how new packages should be implemented. See the detailed documentation for this file.

B.2 Lisp Code Explanations

B.2.1 The Files

This is a list of BOOMS files and a brief description of each. The next sections contain full explanations for the important files — the files marked with a $\star$.

`HEADER` This is a simple header file for the whole system, providing the logo and date on the self-extractor file and on after the system is loaded.

`FILELIST` This is the list of all files, used only for creating the compressed file.

`ARJBOOMS.BAT` This is a small batch-file that creates the compressed file.

`BOOMS.TXT` This is the BOOMS short abstract that is displayed when using the "About Booms" menu command.

`BOOMS.TEX` The source of this work.

`B-FILES.LSP` ⋆ This defines the load and compile mechanisms. It provides 'makefile' mechanisms for reload / recompile when needed. The need for such a file arise as this Lisp version does not contain the `defsystem` facility.

`_LOAD.LSP` When this file is loaded, the whole system is loaded — if this was already done then only modified files will be reloaded.

`_COMPILE.LSP` When this file is loaded, the whole system is recompiled. Only new files will be compiled

`_RUN.LSP` When this file is loaded, the whole BOOMS system is loaded and the application is launched.

`PACKAGES.LSP` ⋆ Definitions for the packages, symbols imports and symbols exports.

`NETWORK.LSP` ⋆ This is the main core of BOOMS. It defines the basic 'Node' class that compose an operational hierarchy DAG.

`BOOMSAPP.LSP` ⋆ The main Windows application definitions.

`BOOMSDOC.LSP` ⋆ The 'Document' class definition. When speaking of the 'Document-View' approach, this file is actually the hierarchical view implementation and `NETWORK.LSP`is the real document.

`ICONS.LSP` ⋆ Definitions for the 'Icon' class which is used to add GUI methods for nodes and create the 'Iconic-Node' class.

`LEAVES.LSP` ⋆ This defines GUI leaf nodes.

`ABSTRACT.LSP` ⋆ Definitions for visual hierarchical abstractions. The actual abstraction mechanism is in `NETWORK.LSP`.

`SHARED-I.LSP` ⋆ The shared icon classes are defined in this file.

`CLASSPAL.LSP` ⋆ The definitions for the icon classes palette, and the mechanism for run-time addition of classes.

`ICONEDIT.LSP` ⋆ This defines the icon edit dialog mechanism, supporting customizable property edit pages.

`COLEDIT.LSP` ⋆ Definitions for the icon color edit pane.

`SAVELOAD.LSP` ⋆ Saving and loading BOOMS documents.

`MUSIC.LSP` ⋆ The music sample package.

`ARITH.LSP` ⋆ The arithmetic sample package.

`GCALC.LSP` ⋆ The implementation of the GCalc double view editor.

`MISC.LSP` ⋆ Miscellaneous useful definitions.

`GR-MISC.LSP` ⋆ Miscellaneous GUI definitions, like the color-box widget.

`GEOMETRY.LSP` ⋆ Geometrical efficient functions.

`CONSTANT.LSP` ⋆ All constants are defined in this file. This can be modified to make the application look different.

`DIALOGS.LSP` ⋆ Several common dialogs that are used in BOOMS are defined here.

`BITMAPS.LSP` ⋆ Bitmaps for the toolbar buttons.

`MIDI-LIB.LSP` ⋆ This is the Lisp interface to the MIDI driver.

`MIDI-LIB.DLL` This is the MIDI driver DLL.

`MIDI-LIB\MIDI-LIB.H` The header file for the MIDI driver library.

`MIDI-LIB\MIDI-LIB.C` ⋆ C functions that implement the simple MIDI driver.

`MIDI-LIB\MIDILIB.MAK` MsDev automatically generated makefile.

`MIDI-LIB\MIDILIB.MDP` MsDev project file.

`ICONS.DLL` Icons that used in BOOMS — for node icons and the window icons.

`LOGO.BMP` The bitmap that pops when BOOMS is started.

`INTRO.WAV` The wave file for the logo pop-up.

`NETWORK.EXA` ⋆ This is an example of using the bare BOOMS core (in `NETWORK.LSP`) with no Graphic User Interface.

Appendix C

The GCalc Mini System

C.1 Scheme Implementation

This system is written in Scheme, the implementation used is John Kozak's[1] Scheme version of SCM (by Aubrey Jaffer[2]). The system was written on top of `bitmap.scm` which is a simple object-oriented extension to the Windows Scheme that makes life simple (written by the author).

This should work for any true 32-bit Windows, but due to some unidentified bug, it works only on some Windows 95 versions (on some, it works but it does not actually draw anything).

C.2 FTPing, Extracting, Running

The GCalc system is at the same FTP site as BOOMS and getting the file is done exactly like getting the BOOMS file, see Section A.2. The file name is `gcalc.exe`, it is also an ARJ self-extractor that should be extracted at the root of the "C" hard-disk (the whole system is about 5Mb).

Running the system is simple, no compiling is needed. The batch file `scm\gcalc.bat` will run the system.

[1] `jkozak@cix.compulink.co.uk`
[2] `jaffer@ai.mit.edu`

C.3 System Documentation

This documentation is from the Scheme file. It is only LaTeX-ized, so it might not follow the overall style of this work.

C.3.1 General Description

The basic description language is composed of simple basic objects and constructors that compose more objects. The word 'object' can mislead the reader to think of a world of objects with identities. This is not the case — and the word object should be read as values that are represented by GUI objects.

- The basic objects are solid colored cubes. The exact colors have no importance, a transparent cube is usually useful.
- All objects are cubes, and composing them using the constructors creates new objects which are also cubes.
- There are three constructors which correspond to the three axes: Left-Right, Top-Down and Front-Back constructions. Each construction takes two objects and produces another — the result is a cube split into two halves containing the two original objects.

This world is perfect for the purpose of demonstrating lambda calculus since it is on one hand very simple, and on the other hand graphic so a visual interface is perfect here.

C.3.2 Functional World

In order to make this description language into a functional language we need to implement functions and the rules for handling them. This is the important step here. Two operations are added — lambda abstraction and application (beta-reduction). Lambda abstraction means taking an expression and making one of its components a variable, and application means taking a lambda abstraction's body and replace each occurrence of the variable by the application argument.

C.3.3 Visualization

The visualization of basic objects and simple constructions of them is straightforward — we use a simple three dimensional cube that is the exact cube to be drawn. The two other cases are a bit more complex.

- Basic cube: Simply draw this cube as a three dimensional object. Note that this can also be the case for variables — we have to draw variable components of expressions as the original object. See explanations below.

- Construction: Draw the two parts of the construction in the two halves defined by the constructor's axis.

- Application: Note that an application does not appear too often, since in most cases it is immediately evaluated (= reduced) — this follows from the evaluation rules below. There are only two cases where an application can 'survive' evaluation — one is the application of a basic cube, which has no meaning except the application itself, so there's nothing to do but to leave this application in place, this can also occur when applying such an application again on another object. The other case can happen as a result of applying some function to an object — if this function is of more than one variable (curried), then the result will be the simple substitution result — no re-evaluation of the body. An application object is visualized as the object being applied behind its argument, in the same dimension size relation so it can't be confused with a front-back construction.

- Lambda Abstraction: An abstraction is simply shown as the body of the abstraction with a little circle which represents the variable object. A lambda abstraction inside another will be drawn as having two circled variables (or more) one next to the other.

  There is something that should be said here on variables — I can't use the substitution model without renaming, so the renaming I use preserve the object that was turned into a variable. This way the variable object can have a color, and the body is still visualized the same. See an explanation below for why the substitution model must be used.

C.3.4 System Description

The system is very simple and follows exactly the principles above. There are three windows — main window and two floating toolbars.

The main window provides the "work-space", it has a cell for the 'selected object' which is being worked on, and eight surrounding cells. Dragging an object to the center cell just put that object instead of what was there before. Dragging an object to the other eight cells performs an operation — three cells create a construction, and three more construct an 'opposite' constructions (switching the first and second arguments). The application cells constructs an application (one way or the opposite) that is immediately being evaluated — all other applied objects are reduced.

The other two panes are the 'Basics' pane and the 'Objects' pane. The Basics pane contains the basic objects that are used to build all compound objects — all color cubes and the transparent cube. This pane cannot be modified and it is created once on startup. An important role of this pane is for constructing lambda abstractions — when a basic object is right clicked the selected object will be abstracted using the basic color clicked as a variable. The Objects pane is a store place to save objects that are needed in the flow of work. Objects can be dragged to the objects pane and back.

Learning how to work with the system is easy — it is a simple, small tool. There are few useful operations that are all specified below.

'H' Shows a help screen that describes these operations.

Left-mouse drag n' drop This is for dragging objects to store locations or to the selected cell. It is also used to drag objects to the action cells.

Right-mouse click Use this to zoom-in an object. All objects are actually drawn on the zoom pane, so viewing a zoomed object means no additional processing. Another important use of a right click (specified above) is constructing a lambda abstraction — this happens when clicking a basic object, which will be used as the variable for the new abstraction. There is also a minor operation for a right button click — on an action cell it will negate the action, for example on the 'top' cell it'll only leave the bottom part etc.

'N' This tends to be very useful for demonstration purposes — it will name the selected object, so objects can have a descriptive name.

'D' This toggles visual drawing on/off. Usually the objects are drawn visually, this helps sometimes understanding the visual object's structure but slows evaluation.

'C' This will clear the selected object. It is useful only for saving files — loading will reevaluate all expressions so it can be useful to delete objects. The cleared cell is such that can be dragged to the Objects pane so other cells can be cleared.

'S' This will save a file with all objects on the Objects pane and the selected cell.

'L' This will load a saved file. Note that loading means reevaluating all objects to draw them, so this operation might take a while — the visual drawing can be turned off for speed.

'R' Restarting the system — clear the Objects pane and the selected cell.

'Esc' Quit.

C.3.5 Building Expressions

The system description above implies that objects are built step by step. This means that the Objects pane must be used from time to time to store intermediate result while building some complex objects. Another important feature of the system is that all applications constructed are immediately evaluated — reduced. One important result of this is that we can't use the extended application rules specified in the Grame paper — the rules that assign functional meaning to basic objects.

These rules give all objects a functional meaning, for example, the red cube can be used as the 'reddish' operator — the application of red on white producing a pink cube. However, using these rules in this system means we cannot construct abstractions that contains applications since these applications will be immediately reduced — when we'll try to build something like "(λ (*red*) (*red white*))", we'll get instead "(λ (*red*) *pink*)". This is because we must construct the red-white application, and we don't have any way of preventing this application from being evaluated. See the "functional extensions" section below for further discussion.

C.3.6 The Evaluation Model

The first question that should be asked when implementing this system is what model will be used for evaluating expressions. There are two immediate options here — the substitution model and the environment model. When I started working on the system I first tried using the environment model — it looks as if it is much more efficient using it.

Using the environment model leads to many problems. The first problem is that our objects should have some kind of a visual representation for objects. It is not a general problem, but it makes other problems get worse since it makes visualization unnatural. Another problem is where should the expressions be evaluated — an abstraction should produce a closure, and this closure should contain some kind of an environment. This comes as a problem considering the nature of this system, that is, the step-by-step construction of objects. When we'll want to construct a two variable lambda abstraction like:

$$(\lambda\ (red)\ \ (\lambda\ (yellow)\ \ (\text{left-right}\ red\ yellow)))$$

we will have to first construct the inner expression — in the global environment, so when later the whole expression is applied, the outer variable will not be bound in the inner one's environment. A possible solution will be to not evaluate lambda abstraction, but then the environment model advantage is lost and the results will not be visualized properly.

The other option is a lot more reasonable here — the substitution model. It is more natural, since this is the way we actually think of using lambda abstractions — what we expect to see as a result of applying an abstraction like "$(\lambda\ (red)\ \ red)$" on "*green*" is indeed "*green*" and not "*red*" in an environment that binds it to "*green*". This means that using the environment model will just be an attempt to simulate substituting with the wrong model. There is only one problem to solve here — avoiding variable capturing, this requires renaming. But — names here are basic colored objects, and they should be later visualized, so the original variable should be kept — so to represent a variable I use a list that contains the variable name and some number. This means that not all information is shown on the screen so confusing situations can arise, like the result of applying

$$(\lambda\ (red)\ \ (\lambda\ (green)\ \ (\text{left-right}\ red\ green)))$$

on green will be visualized like

$$(\lambda\ (green)\ \ (\text{left-right}\ green\ green))$$

but the first green of this construction is a green cube, and the second is a green variable.

C.3.7 Reduction Rules

The last thing (and the most important one) to specify is the reduction rules used. The syntax of the GCalc system is very simple, but there should be an exact specification of the semantics — object evaluation rules.

This is quite simple, and it's best demonstrated by the `eval-exp1` function's simplicity (just 17 lines). Note that this function is used only on an application operation.

Note that these rules do specify the extended reduction rules that gives a meaning to a construction of abstractions. This means that a function can be composed of several parts, and applying such a function will be done by applying each part on the corresponding argument's part. See the rules for exact definition of this.

The rules follow:

Simple expression Just return the expression. These expressions include both basic objects and variable objects.

Construction expressions These expressions are simply evaluated recursively, formally, the expression "(left-right x y)" will be evaluated to "(left-right x' y')" when x' and y' are the evaluations of x and y respectively.

A question can be asked here — is this really needed? Well, in most cases it is not needed, we can simply return the expression as it is. However, there are some rare occasions where the result of an application returns an object which is a construction of applications, such as applying this function on identity (for example, "$(\lambda\ (green)\ \ green))$":

$$(\lambda\ (red)\ \ (\text{left-right}\ (red\ yellow)\ (red\ cyan)))$$

The result of the substitution would be:

$$(\text{left-right}\ ((\lambda\ (green)\ \ green)\ yellow)\ ((\lambda\ (green)\ \ green)\ yellow))$$

But this should be further reduced to:

(left-right *yellow cyan*)

This can happen only as a result of an application, which implies a possible optimization — use a simple version of `eval-exp1` for all expressions, and make the application evaluation use another version that will actually evaluate constructions. This can be simplified by not using `eval-exp1` for any expressions except applications. This is the cause of the conditional call in `eval-exp`.

Lambda Abstraction expressions Abstractions are also returned as they are, these expressions does not have to be evaluated — they are given a variable when first constructed and only change when applied. Note that this case can be considered the same as the construction expressions above — we might want to compute the body of a function as soon as this is possible, however, this is not wanted since the function should be applied again sometimes to get a final result (not a function), so it is not necessary to do the evaluation. This is actually a kind of a normal-order evaluation — where function bodies are not evaluated except when needed. This alone is not enough for normal order evaluation since other rules should do a similar thing (see below).

Application Expressions This is where there is some job to be done. Note that we wouldn't have to use the evaluation function recursively unless, during some applications, there will be more [nested] applications that can be evaluated now. We now split into cases based on the evaluated version of the object being applied:

Simple expressions There's nothing we can do with these expressions, so they are returned as they are (this would be the place to add the extended evaluation rules).

Application expressions This is also treated as a simple expression with nothing much to be done, since the expression is anyway something that is not a simple (non-function) object.

These expressions could be further evaluated in case of a substitution, and not doing this creates the normal order effect — see the rule for reducing lambda abstractions above for further explanations. For example, this enables us to enter this expression

without getting into an infinite loop (however, evaluating its application on a function will get us into the unavoidable loop):

$(\lambda\ (red)\ \ (\lambda\ (blue)\ (red\ (blue\ blue)))$
$\qquad\qquad (\lambda\ (blue)\ (red\ (blue\ blue))))$

Abstraction expressions Here we replace the application argument for the abstraction variable in the abstraction body, and return the evaluation of this substitution result (calling the evaluation function recursively).

Construction expressions This is the special extension of a function object. This extension gives functional meaning to constructions of functions. What is being done here is applying a function part on the corresponding argument's part. For example, the result of this application:

$$((\text{left-right}\ f_1\ f_2)\ obj)$$

would be:

$$(\text{left-right}\ (f_1\ (\text{left-part}\ obj))(f_2\ (\text{right-part}\ obj)))$$

And now the formal generalization of this example should be obvious now. This operation must be done only when the object is constructed in the correct direction, therefore, it is used on the result of a split function that will make the object a construct in the correct size (splitting it if it is not already split in the correct direction). Note that this evaluation rule is really powerful — it enables the usage of "cubes of functions" on other cubes. There is little meaning to cubes of abstractions with different "lambda lengths" (objects like half functions and half basic colors), but there is no error checking.

[Last remark — following this rule exactly explains why the Grame's "Spenger Sponge" won't work as shown in their paper, see the last section below.]

C.3.8 Functional Extensions

Now that we've seen the GCalc definition we can spend a thought on the extensions. The one extension I have implemented is the construction of functions, which is taken as a function that does something on one half and

another on the other half. This is quite natural to understand, and very useful when using the system for tricky demonstrations. This extension makes the GCalc world more consistent in the sense that the new objects introduced — abstraction and application, can be taken like all other objects, so it is possible to use them with constructions.

The Grame paper specifies another extension — we have our new objects behave like any other object by now, what we want is the other way — have simple objects behave like functions when they are applied. This was not done in Grame's system as well as in this system for two main reasons:

1. Implementing this means there should be a mechanism that will specify when we actually want to evaluate an expression. This is because (as already said above) the expressions are built piece by piece from the inside parts to the outside. This will prevent us from building some expressions as specified above ("building expressions" section). A solution for this will very easy to implement, simply have some event like a key-press or double-click evaluate the selected object, but it'll make the system somewhat too complex to be fun — keeping in mind that expressions should get evaluated sometime.

2. This meaning of basic objects as functions (red as a reddish function etc.) is the most natural choice, however, it is not natural enough to make a user mad about not having this feature.

So, to conclude, this extension was not implemented for the sake of keeping this a "toy system" that will be fun to play with. This has caused the evaluation function to look a bit messy (when trying to dig inside it and see what do get evaluated and what not — see the above section), but the result is worth this. A serious implementation would have this feature, but then again — what serious things can be done with this cube world?

C.3.9 Formal Definition of GCalc

The formal definition of the GCalc system is given at Sections 4.3, 5.2.1 and 5.2.3.

Bibliography

[1] Felix Salzer. *Structural Hearing: Tonal Coherence in Music*. Dover Publications, 1962.

[2] E. Narmour. *Beyond Schenkerism: The Need for Alternatives in Music Analysis*. The University of Chicago Press, 1977.

[3] Fred Lerdahl and Ray Jackendoff. *A Generative Theory of Tonal Music*. The MIT Press, 1983.

[4] Terry Winograd. Linguistics and the computer analysis of tonal harmony. *Journal of Music Theory*, 12:2–49, 1968.

[5] O. Laske. Introduction to a generative theory of music. *Sonological Reports*, 16, 1975.

[6] O. Laske. Composition theory: An enrichment of music theory. *INTERFACE, Special Issue on Models of Musical Communication and Cognition*, 18(1–2):45–59, 1989.

[7] S.W. Smoliar. An approach to music theory through computational linguistics. *Journal of Music Theory*, pages 105–131, 1976.

[8] M. Kassler. Explication of the middleground of schenker's theory of tonality. *Miscellanea Musicologica*, 9:72–81, 1977.

[9] A.X. Rodet and P. Cointe. Formes: Composition and scheduling of processes. *Computer Music Journal*, 8(3):32–50, 1984.

[10] W. Buxton, W. Reeves, R. Baecker, and L. Mezei. The use of hierarchy and instance in a data structure for computer music. *Computer Music Journal*, 2(4):10–20, 1978.

[11] R.B. Dannenberg, P. McAvinney, and D. Rubine. Arctic: A functional language for real time systems. *Computer Music Journal*, 10(4):67–78, 1986.

[12] Mira Balaban. The music structures approach to knowledge representation for music processing. *Computer Music Journal*, 20(2):96–111, 1996.

[13] D.V. Oppenheim. Dmix: A multi faceted environment for composition and performing computer music: Its design, philosophy, and implementation. In *Computer Music Days*, pages 1–10, Delphi, Greece, 1992.

[14] S. Letz, Merlier, and Y. Orlarey. Common lisp compositional environment : Manuel de reference. Technical report, GRAME, Lyon, 1990.

[15] Mira Balaban and Yan Orlarey. Score ontology: A step towards constructing a formal semantics for computer music tools. Technical report, Department of Industrial Engineering and Management, Ben-Gurion University, Beer Sheva and Department of Musicology, Bar-Ilan University, Ramat Gan, Israel, and GRAME, Lyon, France, 1996.

[16] J. Gourlay. A language for music printing. *Communications of the ACM*, 29:388–401, 1986.

[17] G. Loy. Musicians make a standard: The midi phenomenon. *Computer Music Journal*, 9(4):8–26, 1985.

[18] Keith McMillen. Zipi: Origins and motivations. *Computer Music Journal*, 18(4):47–51, 1994.

[19] Y. Orlarey, D. Fober, S. Letz, and M. Bilton. Lambda calculus and music calculi. In *International Computer Music Conference*. San Francisco: International Computer Music Association, 1994.

[20] H. Taube. Stella: Persistent score representation and score editing in common music. *Computer Music Journal*, 17(4):38–50, 1993.

[21] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. Addison-Wesley Publishing Company, 1990.

[22] D.J. Kurlander. *Graphical Histories*. PhD thesis, Columbia University, Computer Science, 1992.

[23] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1986.

[24] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *Human Factors In Computing Systems*, pages 83–92, 1992.

[25] Krasner G.E 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, 1988.

[26] Ben Shneiderman. *Designing the User Interface, Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publishing Company, second edition, 1992.

[27] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–529, 1988.

[28] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, 2nd edition, 1989.

[29] M. Balaban and C. Samoun. Hierarchy, time and inheritance in music modelling. *Langugaes of Design*, 1(2):147–172, 1993.