

# Abstraction as a Means for End-User Computing in Creative Applications

Mira Balaban (mira@cs.bgu.ac.il)

Eli Barzilay (eli@cs.cornell.edu)

Michael Elhadad (elhadad@cs.bgu.ac.il)

**Abstract**—End-User computing is needed in creative artistic applications or integrated editing environments, where the activity cannot be planned in advance. Following [1], *Concrete abstractions* (abstractions from examples), are suggested as a new mode for function definition, appropriate for end-user editor programmability. For certain applications, the direct, associative, not planned in advance character of concrete abstraction plays a qualitative role in the mere ability to specify abstractions.

In this paper we propose to use concrete abstraction as a general tool for end-user programmability in editors. We distinguish two kinds of abstractions: *value abstraction* and *structure abstraction*, and explain how they can be combined. We describe a framework of historical editing that is based on a double view, in which the two abstraction kinds are combined. Finally, *BOOMS* [2], an implemented prototype for such an editing framework is described. *BOOMS* is a domain independent toolkit, with three sample instantiations. We believe that the proposed framework captures the conceptualization operation that characterizes creative, associative work types, and addresses the needs for end-user computing in integrated environments.

**Index Terms**—Concrete abstraction, creative applications, music composition, end-user computing, historical editing, integrated environments.

## I. INTRODUCTION

**E**ND-USER computing is needed in domains or applications where the activity cannot be planned in advance. For example, in artistic applications, an experimental mode of operation is common. In that mode the artist “plays with the material” until the “right” intentions are formed. In music composition a composer might wish to abstract away some parameters from a concrete piece, generalize the structure of a piece, apply these patterns on a different material, combine and repeat patterns, etc. These methods have the quality of programming processes, even if they are not usually termed as such.

Smart editors also provide capabilities for end-user computing, since the designer of the editor cannot foresee and prepare procedures for the desired patterns of editing and of user behavior. Modern editors function as general computer environments that control the overall range of activities involved in a human-computer interaction. The editor is the management system that supports generation, update, modification, testing, running applications, providing feedback, system integration, etc. As such, modern editors need to provide services that

go beyond the immediate command-reaction character of traditional editors. These objectives are achieved by enhancing editors with *end-user programming capabilities*.

Editor programmability can help in avoiding repetitive operations, and ensure consistency by abstracting complex operations. In a text editor, for example, a user might wish to make all emphasized text use a bold font instead of italic font. In a graphical editor, a user might wish to change the background color of all square windows to blue. These are examples for *simple* operations that should be applied to *many* objects. An example for a complex operation that should be applied to multiple objects is: “make all grids that represent numerical tables use double lines.” Although this operation might involve only few objects, a uniform application via abstraction can help in preventing user errors that can be expected when complex operations are repeated.

Some editors are further strengthened to include full programming capabilities. These include the ability to combine primitive editor operations to form compound structures of editor operations, create editor functions (*abstractions*) that apply editor operations to document arguments, and enable *naming*. For example, *Emacs* uses a full programming language (Emacs Lisp), enhanced with document data structures and primitive editor operations, that are integrated within a user interface; *Word* enables users to define macro operations, and write programs in Word Basic which is a variant of Visual Basic; and graphic tools such as *Photoshop* can create HTML widgets.

Nevertheless, standard end-user computing requires programming capabilities on the part of the user. In artistic applications, like music composition, as described above, planned end-user programming is not relevant, since the activity is not planned and the users are not programmers. In powerful editors planned end-user programming is also unsatisfactory, even for users that have programming capabilities. The reason is that most users are occupied with the subject matter of their application, and are not willing to devote the time needed for programming their editing environment. Some Lisp programmers, indeed, bother to program and personalize their *Emacs* environments, but most naive users simply repeat their operations, and sometimes even do not notice the possibility of abstraction.

The conclusion is that good end-user computing should have the flavor of “on-the-fly” computing, *i.e.*, should emerge during the activity itself, when the user desires to create a combination/repetition/abstraction/naming construct, based

on some concrete material. *Concrete abstraction* (abstraction from concrete examples), was first suggested by Yann Orclarey and his colleagues in [1], as a form of end-user editor programmability that is essential for music composition. The idea is, roughly, to provide users with the capability to abstract a concrete music piece into a pattern, and then apply the pattern to other music pieces, thus yielding new music pieces, all admitting the same pattern. Concrete abstraction turns out to be an extremely powerful end-user programmability means in music, since the abstraction can be applied to different music parameters. The recent *Elody* composition environment built in GRAME [3], [4] is centered around the concrete abstraction user operation. A special case of concrete abstraction is employed, in an implicit manner, in the *DMIX* real time music composition environment of Oppenheim [5]. Using *DMIX*, a composer can create a music piece, abstract its rhythmic structure away from it, and then “slap” another rhythmic structure on it. This way, Oppenheim creates examples of “Jazzified Bach preludes.”

Concrete abstraction is a method of function definition that is generally not supported by programming languages. Programmers define functions in a planned and thoughtful mode: they (analytically) observe the existence of a possibly useful abstraction and use special linguistic symbols for variables rather than using world objects (or their representations) directly. Programmable editors do not generally support concrete abstraction either. The novelty of concrete abstraction lies in the new mode for function definition. For certain applications, as well as for programmable editors, the direct, associative, not planned in advance character of concrete abstraction plays a qualitative role in the mere ability to specify abstractions.

In this paper we propose to use concrete abstraction as a general tool for end-user programmability in editors. We distinguish two kinds of abstractions: *value abstraction* and *structure abstraction*, and explain how they can be combined. We describe a framework of historical editing that is based on a double view, in which the two abstraction kinds are combined. Finally, *BOOMS* [2], an implemented prototype for such an editing framework is described. *BOOMS* is a domain-independent toolkit, with three sample instantiations. We believe that the proposed framework captures the conceptualization operation that characterizes creative, associative work types, and addresses the needs for end-user computing in integrated environments.

Section II briefly introduces abstraction in Lambda Calculus. Section III defines value and structure abstractions. Sections IV and V describe the combination of the two abstraction kinds, and the historical editing framework. Section VI describes related work and Section VII is the conclusion and future work. Appendix I gives a brief overview of the *BOOMS* system.

## II. BACKGROUND ON ABSTRACTION

Abstraction is an act of generalization. For example, abstracting the red color from a red flower gives the generalized concept of a colorful flower, whose color is unknown. Consider the motif in the following Example.

Example 1:



We represent a note as a pair: ( $\langle \text{pitch} \rangle$ ,  $\langle \text{duration} \rangle$ ). In order to keep with the traditional tonal terminology, a pitch specifier consists of a diatonic name and an octave specifier, such as  $C_0$  for  $C$  in the middle octave, or  $C_{5-5}$  to specify the same thing using a numerical expression for the octave specifier. Note sequencing (sequential concatenation), is represented by the ‘-’ operation. The symbolic notation for the overall motif is:

$$(C_0, \frac{1}{4}) - (E_0, \frac{1}{4}) - (F\sharp_0, \frac{1}{4}) - (E_0, \frac{1}{8}) - (E_0, \frac{1}{8}) - (D\sharp_0, 1) \quad (1)$$

Abstracting this motif over the pitch ‘ $E_0$ ’ yields the pattern

$$(C_0, \frac{1}{4}) - (\square, \frac{1}{4}) - (F\sharp_0, \frac{1}{4}) - (\square, \frac{1}{8}) - (\square, \frac{1}{8}) - (D\sharp_0, 1) \quad (2)$$

that functions as a *motif generator*: The “hole”  $\square$  can be replaced by anything that evaluates to a pitch. The motif pattern can be further abstracted, for example over the ‘ $\frac{1}{4}$ ’ duration, to yield a new motif generator, with two kinds of holes:

$$(C_0, \Delta) - (\square, \Delta) - (F\sharp_0, \Delta) - (\square, \frac{1}{8}) - (\square, \frac{1}{8}) - (D\sharp_0, 1) \quad (3)$$

The “hole”  $\Delta$  can be replaced by anything that evaluates to a duration. The *mode of combination* itself can also be a subject for generalization, yielding a new motif generator, with three kinds of holes:

$$(C_0, \Delta) \circ (\square, \Delta) \circ (F\sharp_0, \Delta) \circ (\square, \frac{1}{8}) \circ (\square, \frac{1}{8}) \circ (D\sharp_0, 1) \quad (4)$$

The “hole”  $\circ$  can be replaced by anything that evaluates to a motif combination operator.

Application is the opposite operation — an act of specialization of a generalized concept by means of substitutions. For example, replacing  $\Delta$  by  $\frac{1}{2}$ ,  $\circ$  by “concatenate within a delay of  $\frac{1}{8}$ ” (denoted  $\parallel_{\frac{1}{8}}$ ), and  $\square$  by  $\mathcal{G}_0 - \mathcal{E}_0$ , yields the following motif:

Example 2: The symbolic representation of this motif is:



$$(C_0, \frac{1}{2}) \parallel_{\frac{1}{8}} [(\mathcal{G}_0, \frac{1}{4}) - (E_0, \frac{1}{4})] \parallel_{\frac{1}{8}} (F\sharp_0, \frac{1}{2}) \parallel_{\frac{1}{8}} \quad (5)$$

$$[(\mathcal{G}_0, \frac{1}{16}) - (E_0, \frac{1}{16})] \parallel_{\frac{1}{8}} [(\mathcal{G}_0, \frac{1}{16}) - (E_0, \frac{1}{16})] \parallel_{\frac{1}{8}}$$

$$(D\sharp_0, 1)$$

Note that the computation of this application requires evaluation rules for structured pitch specifiers (expressions). The straightforward replacement, dictated by the recent application yields the “motif”:

$$(C_0, \frac{1}{2}) \parallel_{\frac{1}{8}} (\mathcal{G}_0 - \mathcal{E}_0, \frac{1}{2}) \parallel_{\frac{1}{8}} (F\sharp_0, \frac{1}{2}) \parallel_{\frac{1}{8}} \quad (6)$$

$$(\mathcal{G}_0 - \mathcal{E}_0, \frac{1}{8}) \parallel_{\frac{1}{8}} (\mathcal{G}_0 - \mathcal{E}_0, \frac{1}{8}) \parallel_{\frac{1}{8}} (D\sharp_0, 1)$$

To obtain a real motif, we still need to compute the intended note meaning of  $(\mathcal{G}_0 - \mathcal{E}_0, \frac{1}{2})$  and  $(\mathcal{G}_0 - \mathcal{E}_0, \frac{1}{8})$ . A reasonable rule is: “Replace any  $(N_1 - \dots - N_m, D)$  expression by the note sequence:  $(N_1, \frac{D}{m}) - \dots - (N_m, \frac{D}{m})$ .” Applying this rule for the recent application yields the motif in Example 2.

To conclude:

**Abstraction** is an operation 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  $v$  from some simple value  $a$  means “stripping off” the  $a$  property from  $v$ , creating a generalized object — a function to be applied later. Technically, the result of such an abstraction is replacing each occurrence of  $a$  in  $v$  by a variable  $x$ , yielding a function of a single parameter  $x$ .

**Application** is the opposite operation — instantiating an abstraction. It takes an abstraction function  $f$  with parameter  $x$ , and some value  $v$ , and instantiates (replaces) the abstraction parameter by the applied value. So, while the abstraction operation creates a function  $f$  with a parameter  $x$ , the application operation results in an application of the abstraction  $f$  on a given value  $v$ . For example, application of the above colorful flower on a yellow color yields a yellow flower. This gets us closer to the very basic notion of Lambda abstraction as defined in Lambda Calculus, where abstraction on values is the only means for function creation.

#### A. Lambda Calculus: Abstraction & Application Formalism

Lambda Calculus [6], [7] is a simple language of expressions that are generated by *juxta-positioning* and *Lambda abstractions*. The intended meaning of a Lambda abstraction construct is a function based on a given pattern (expression). The intended meaning of juxta-positioning is that of application. Overall, there are three kinds of expressions<sup>1</sup>:

- 1) Symbols (atoms);
- 2) Applications:  $\langle expression \rangle \langle expression \rangle$ ;
- 3) Lambda expressions:  $\lambda \langle variable \rangle . \langle expression \rangle$ .

Parenthesizing is used for resolving ambiguities. We take as given some built-in arithmetic primitives (such as numerals and simple operators).

For example, the expression  $(+ n n)$  can be abstracted into the “ $n+n$ ” function  $\lambda n. (+ n n)$ , and then further abstracted into the “ $n$  op  $n$ ” higher order function  $\lambda op. \lambda n. (op n n)$ . Computation in this calculus is obtained by *application* of  $\lambda$ -expressions to their right neighbor expressions, taken as arguments. This application is called *reduction*, and it is done by means of *substitution*. Repeating these reductions as much as possible is called *evaluation*. Here are some examples:

- The application  $((\lambda n. (+ n n)) 2)$  reduces to  $(+ 2 2)$ , which yields ‘4’. Similarly,  $(+ (\lambda n. (+ n n) 2) 3)$  evaluates to ‘7’.
- The application  $((\lambda op. \lambda n. (op n n)) * 2)$  reduces into  $((\lambda n. (* n n)) 2)$  which yields ‘4’.

<sup>1</sup>The following explanation is highly informal. For a complete and formal description the reader is referred to [6], [7].

- The application  $((\lambda f. (f * 2)) (\lambda op. \lambda n. (op n n)))$  reduces into  $((\lambda op. \lambda n. (op n n)) * 2)$ , then it reduces to  $((\lambda n. (* n n)) * 2)$ , then  $(* 2 2)$  which finally yields ‘8’.

Note that it is only possible (using the rules given) to create functions of one variable. The way to imitate  $N$ -ary functions is to abstract one-by-one on all variables, getting a function that returns a function. To apply this function, we again apply it on all inputs, one-by-one. This is called *currying*. For the purpose of this paper,  $\lambda x. \lambda y. \dots$  can be regarded as a simple, two-variable function.

The Lambda Calculus is a full computational model, having Turing Machine equivalent power. It provides the basis for functional programming languages like Scheme, Lisp and ML. Our interest in Lambda Calculus here results from the explicit support for the operations of *concrete abstraction*, and *application*. This observation was first made by Yann Orlarey from GRAME<sup>2</sup>. For example, the Motif generator examples from the introduction can be formalized in Lambda Calculus, as follows:

Motif generator 2:

$$\lambda \mathcal{E}_0. ((\mathcal{C}_0, \frac{1}{4}) - (\mathcal{E}_0, \frac{1}{4}) - (\mathcal{F}_{\#0}, \frac{1}{4}) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{D}_{\#0}, 1))$$

Motif generator 3:

$$\lambda \frac{1}{4}. \lambda \mathcal{E}_0. ((\mathcal{C}_0, \frac{1}{4}) - (\mathcal{E}_0, \frac{1}{4}) - (\mathcal{F}_{\#0}, \frac{1}{4}) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{D}_{\#0}, 1))$$

Motif generator 4:

$$\lambda -. \lambda \frac{1}{4}. \lambda \mathcal{E}_0. ((\mathcal{C}_0, \frac{1}{4}) - (\mathcal{E}_0, \frac{1}{4}) - (\mathcal{F}_{\#0}, \frac{1}{4}) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{D}_{\#0}, 1))$$

In the pure Lambda Calculus, there is a uniform pool of symbols. Hence, abstracting on ‘ $\mathcal{E}_0$ ’, on ‘ $\frac{1}{4}$ ’, and on ‘ $-$ ’ are all the same. However, in realistic implementations, like Scheme and Lisp, atoms are distinguished into different types. In particular, there are constants (that evaluate to themselves) and variables. For example, numbers and strings are constants in Scheme and in Lisp. In such contexts, abstracting on a constant involves substituting its occurrences by a variable, since a constant cannot play the role of a variable (cannot evaluate to anything which is not its self-evaluating value). For example, motif generator 4 used the constant ‘ $\frac{1}{4}$ ’ as a variable name<sup>3</sup>, so we replace it by a variable and we get:

$$\lambda -. \lambda dur. \lambda \mathcal{E}_0. ((\mathcal{C}_0, dur) - (\mathcal{E}_0, dur) - (\mathcal{F}_{\#0}, dur) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{D}_{\#0}, 1))$$

The application of the last motif generator to  $\mathcal{G}_0 - \mathcal{E}_0$  as an ‘ $\mathcal{E}_0$ ’, to ‘ $\frac{1}{2}$ ’ as the *dur*-ation, and to  $\frac{1}{8}$  as an ‘ $-$ ’, that yields motif 5, is obtained by evaluating the application expression:

Motif generator 5:

$$\begin{aligned} & (\lambda -. \lambda dur. \lambda \mathcal{E}_0. \\ & ((\mathcal{C}_0, dur) - (\mathcal{E}_0, dur) - (\mathcal{F}_{\#0}, dur) - \\ & (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{D}_{\#0}, 1))) \\ & \frac{1}{8} \frac{1}{2} (\mathcal{G}_0 - \mathcal{E}_0) \end{aligned}$$

<sup>2</sup>Orlarey further points to the artistic importance of keeping the names of the original values, and not replacing them by synthetic variable names.

<sup>3</sup>In these languages, ‘ $-$ ’ and ‘ $\mathcal{E}_0$ ’ are valid variable symbols. ‘ $\frac{1}{4}$ ’ is not, since it is a numeric constant.

and applying the simplification rule:

$$(N_1 - \dots - N_m, D) \rightarrow (N_1, \frac{D}{m}) - \dots - (N_m, \frac{D}{m})$$

Using Lambda Calculus as an underlying theory for a concrete abstraction based editor, we guarantee the correctness of the implementation semantics, since all the theoretic machinery involving scoping, naming, and reduction order is well-understood. In the rest of this paper, we will use an intuitively relaxed Lambda Calculus notation for examples.

### III. AN EDITOR WITH CONCRETE ABSTRACTION: VALUE AND STRUCTURE

Existing editors can be classified into “what-you-see-is-what-you-get” editors, and “specification-based” editors. The first kind can be termed *extensional editors*<sup>4</sup>, since they apply editing operations directly to the end product, *i.e.*, extensional document. For example, in a “numbering environment” in *Word*, the operation “insert end-of-line” is interpreted as adding a new numbered entry. That is, the user applies the editing operation directly to the numbered text (extensional), and the editor interprets it and performs it, based on the operation and the context (properties of the document). The second kind of editors can be termed *intensional editors*<sup>5</sup>, since they apply editing operations to an intended specification of the end product, rather than to the extensional product itself. For example, *Emacs* and  $\LaTeX$ -mode is an intensional editor, since the editing operations are applied to an intensional-structured specification of the extensional document.

The two kinds of editors have complementary properties. Work with an extensional editor is intuitive and immediate, but it is up to the editor to interpret the user intention, and up to the user to guess the editor’s interpretation. Work with an intensional editor is, usually, more demanding, since the intensional specification might be complex to understand, and the user might be confused about the exact extension being specified. Yet, intensional specification is richer, and might express fine differences that may be cluttered in an extensional end-product. For example, a structured visual object, whose components can be combined by putting them one next to the other, or behind each other, or on top of each other, might be constructed in different orders, all yielding the same object. An intensional editor can capture such differences, while an extensional one cannot. Moreover, there might be intensional relationships among the components of such an object, *e.g.*, specifying that the edge components must be the same. Such a specification means that the replacement of a component object on one edge implies also the automatic replacement of the component on the other edge. An extensional editor cannot capture such distinctions and relationships.

The addition of concrete abstraction to an editor raises the question as to the kind of the editor. Since concrete abstraction is applied directly to the object on which the editor operates, it is not surprising that the two kinds of editors give rise to two different kinds of concrete abstraction.

<sup>4</sup>Following the Logic terminology, where the actual denotation of a symbol in the real world is called its *extension*.

<sup>5</sup>Following the Artificial Intelligence terminology, where the specification of objects or events in the real world is termed *intension*.

Extending an extensional editor with concrete abstraction can be termed *extensional abstraction*. It was first introduced and implemented by Yann Orlarey and his colleagues in [1]<sup>6</sup>. Extending an intensional editor with concrete abstraction can be termed *intensional abstraction*. Intensional abstraction can be further refined into *value abstraction* and *structure abstraction*. These distinctions were first introduced by Eli Barzilay in his *BOOMS* system [2].

Value abstraction is *value based*, *i.e.*, it is applied to all occurrences of a value that appears in the edited object. All music examples described earlier demonstrate value abstractions, since they abstract all occurrences of the note ‘ $\mathcal{E}_0$ ’, or of the duration ‘ $\frac{1}{4}$ ’, or of the operation ‘ $-$ ’. The abstraction supported by the Lambda calculus is a value abstraction. Value abstraction can be used in an extensional and in an intensional editor. The *Elody* composition environment built recently by the GRAME group [3], [4] is an intensional environment with a value concrete abstraction. That means that a user of the *Elody* environment can create a music object, *e.g.*, an ‘ $A - (A+2) - A$ ’ melodic construct of three notes, where  $A$  is a note, and ‘ $A+2$ ’ is a note 2 half tones higher than  $A$ . The user can then apply concrete value abstraction on the  $A$  note, yielding the melodic three-element pattern of ‘ $\lambda A. A - (A+2) - A$ ’. In an extensional editor, the music object ‘ $A - (A+2) - A$ ’ would first be computed (extended), resulting, lets say, ‘ $A - B - A$ ’. The value abstraction on ‘ $A$ ’ would result in ‘ $\lambda A. A - B - A$ ’, thereby suppressing the ‘ $A+2$ ’ intension for ‘ $B$ ’.

Structure abstraction is *structure based*, *i.e.*, it is applied to references to structural components of the, necessarily intensional, edited object. For example, in the above ‘ $A - (A+2) - A$ ’ melodic example, the structure can either identify the first and last occurrences of ‘ $A$ ’ using a single reference, or distinguish them, using two different references. In the first case, structure abstraction on the single reference to ‘ $A$ ’ yields the same pattern as value abstraction on ‘ $A$ ’, while in the latter case, structure abstraction on the reference to the first ‘ $A$ ’ yields a different pattern: ‘ $\lambda ref. ref - (ref+2) - A$ ’ (assuming that the second component uses reference to the first ‘ $A$ ’).

In this section we introduce these two kinds of concrete abstraction. Value abstraction is introduced by observing the GCalc tool of [1] for creative experimentation with colored cubes. Structure abstraction is introduced using synthetic examples that are inspired by the *BOOMS* system.

#### A. GCalc: A Value Abstraction Based Editor

The GCalc system was motivated by the wish to endow a music composition environment with conceptualization capabilities. The idea is to use on the fly creation of abstractions in an intuitive way. In order to explore these ideas, the authors concentrated on a simple structured domain of colored cubes, where they demonstrate that adding value abstraction can produce surprisingly complex results.

The domain of GCalc consists of structured colored cubes. The atomic values are cube colors: red, green, blue, white, transparent and so on. Structured values are created with three

<sup>6</sup>Although they did not use this terminology.

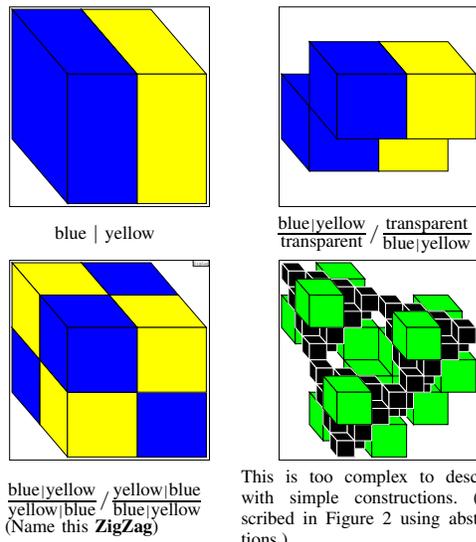


Fig. 1

## SOME STRUCTURED CUBE EXAMPLES

constructors: Left-Right (**LR**), Top-Bottom (**TB**) and Front-Back (**FB**). The structured cubes of GCalc are pure data values, since they carry no identifying states such as their grid locations. Sameness in GCalc is pure value equality. Figure 1 presents several structured cubes<sup>7</sup>. In the example, we use the notation of [1] for the operators: **LR** is denoted  $\square|\square$ , **TB** is denoted  $\frac{\square}{\square}$  and **FB** is denoted  $\square/\square$ . Additional examples, emphasizing the power of concrete abstraction, and the full implementation are described in [2].

Adding concrete value-abstraction and application to the GCalc editor results in an editor that can create and apply, on the fly, single argument functions formed by color abstractions. Figure 2 presents a GCalc running session, with some concrete value-abstraction and application examples. Abstraction and application are conceived as two new constructors: the ‘abstract’ operation creates function (lambda) expressions, and the ‘apply’ operation creates application (reduction) expressions. Application expressions are evaluated after their creation, yielding possibly new expressions. In that sense, GCalc is an *extensional editor* with concrete *value-abstraction*. The addition of abstraction yields a new kind of atomic values — variables (based on colored cubes).

### B. Domain Specific Observability and Sameness

The values of a structured domain are formed with domain constructors like the **LR**, **TB**, and **FB** constructors in the colored cubes domain. In most domains the constructors obey certain regulations with respect to the domain specific observation means. Consequently, different structured values appear to be the same (*i.e.*, they have the same *normal form* with respect to the common observation means; they “evaluate visually” to the same data value). For example, if a constructor “ $\circ$ ” is observed to be idempotent and commutative, then  $(v \circ v)$

<sup>7</sup>The example is based on a Scheme implementation for GCalc, done by Eli Barzilay as part of the *BOOMS* project.

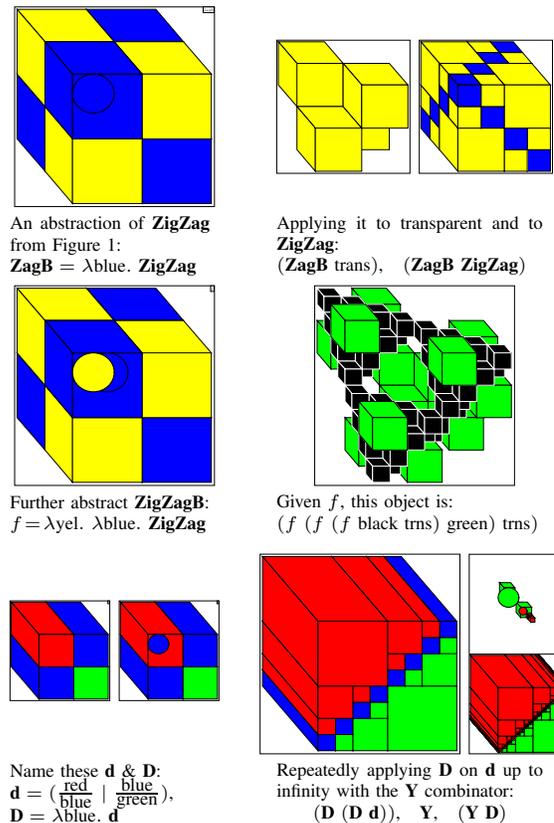


Fig. 2

## EXAMPLES OF ABSTRACTIONS AND APPLICATIONS

is observably the same as  $v$ , and  $(v_1 \circ v_2)$  is observably the same as  $(v_2 \circ v_1)$ . We demonstrate the problematic nature of observable dependent equality in the two domains governing this paper, the cubes and the music domains.

**The cubes domain:** Since the obvious observation means is the cubes visualization, the structured cubes  $\left( \frac{\text{red}|\text{green}}{\text{green}|\text{red}} \right)$ , and  $\left( \frac{\text{red}}{\text{green}} \mid \frac{\text{green}}{\text{red}} \right)$  have exactly the same visualization: . Similarly,  $(\text{red} \mid \text{red})$ ,  $\left( \frac{\text{red}}{\text{red}} \right)$ ,  $\left( \frac{\text{red}|\text{red}}{\text{red}} \right)$ , etc., have the same visualization, a red cube.

**The music domain:** The obvious, but not necessarily musically correct, observation means is the music piece, *i.e.*, the timed set of notes denoted by a music specifier. With respect to this observable, if  $d_1$  and  $d_2$  are duration expressions that evaluate to the same duration, then  $(p, d_1)$  and  $(p, d_2)$  denote the same note, using a note constructor that inserts some default values for the dynamics and the timbre parameters. Similarly,  $((p, d) \mid (p, d))$ <sup>8</sup> is observably the same as  $(p, d)$ . Likewise, if  $M_1, M_2, M_3, M_4$  are motif values, then  $((M_1 \mid M_2) - (M_3 \mid M_4))$  and  $((M_1 - M_3) \mid (M_2 - M_4))$  are observably the *same*, although they carry *different* music intentions.

We see that *observable sameness* of data values in a domain depends on the available observation means and on domain specific properties of the constructors. Since concrete value

<sup>8</sup>The operation of note simultaneity is denoted by “ $\mid$ ”.

abstraction as defined by the Lambda Calculus is domain independent, it seems rational for an abstraction based editor to keep the abstraction engine independent from domain rules and from observation means that presumably will constantly be refined (thereby turning previously same values different). The editor might assume that each specific domain provides a *total* and *efficiently computable* normalization procedure for its values<sup>9</sup>.

The normalization procedure should be distinguished from the *observable-rendering* procedure that is used to actually present the data values to the user. Using an observable-rendering procedure allows the editor to maintain objects that are observed the same but represent different intentions. In the cubes domain, the observable-rendering procedure is the graphical rendering procedure that visualizes a structured cube, based on the properties of the cube constructors. In the music domain, the `play` procedure, that computes the actual timed set of notes, based on the properties of the music constructors, serves as an observable-rendering procedure.

*Sameness and intensionality:* Consider a concrete arithmetic expression like  $2 \times 2$ . A user might wish to express explicitly that 2 is to be multiplied by *itself*, (e.g., expressing the area of a square), or by another number that, incidentally, happens to be 2 (e.g., a 2 by 2 rectangle). The printed form looks similar in both cases, but the intended meaning is different — a programmer would use *multiplication* for the second case and *square* for the first. Using planned abstraction, the two intensions are captured by the Lambda expressions:

- $((\lambda x. x \times x) 2)$
- $((\lambda x. \lambda y. x \times y) 2 2)$

This distinction between observability and sameness does not arise from sameness algebraic properties of domain constructors, but from different intensions carried by components of the expression. This distinction, that the user is interested in preserving, cannot be captured by concrete intensional value abstraction. It is precisely for that purpose that we introduce concrete structure abstraction.

### C. Structure Abstraction

The dual structure-intensional companion of concrete value abstraction is concrete structure abstraction. Value abstraction is applied to uniform data values, while structure abstraction is applied to intensional objects that carry, in addition to their value, an identity. Consider, for example, the “Three Men” picture in Figure 3. Suppose we edit this picture using an editor that supports concrete abstractions; if we try to abstract on the hair colors in this picture we get something like  $\lambda\text{black. } \lambda\text{grey. OriginalPicture}$ .

However, suppose we realize that the hair and beard color of the third person are intensionally identified, while it is just a coincidence that the first person has the same hair color. Accordingly, an intension-aware abstraction of the color pattern in this picture should separate the abstraction on the hair

<sup>9</sup>In the GCalc implementations [1], [2] the immediate escape from normalization takes the identity function as a normalization procedure. As a result, GCalc cannot identify different values with the same observable, as equal values.



Fig. 3  
THE “THREE MEN” PICTURE

color of the first person from the abstraction on the necessarily identical color of the hair and beard of the third. This is impossible with value abstraction since all black regions have identical color. In order to implement this intension we need to conceive each color occurrence as an object, and identify the hair and beard color of the third person as the same color object. The intended abstraction should be applied to the reference to that color object. In a Scheme-like language, this structured view can be written:

```
(let ((color1 'black)
      (color2 'red)
      (color3 'black))
  (picture-of-men
   (make-person1 'hair color1)
   (make-person2 'hair color2)
   (make-person3 'hair color3
                  'beard color3)))
```

and the desired abstraction is:

```
(lambda (color1 color2 color3)
  (picture-of-men
   (make-person1 'hair color1)
   (make-person2 'hair color2)
   (make-person3 'hair color3
                  'beard color3)))
```

Name this abstraction “The3Men.” Applying it to the color values “white”, “white”, “green”, i.e.,  $(\text{The3Men } 'white 'white 'green)$ , yields a correct answer.

Structure abstraction in the context of the cubes domain can allow selection of particular color occurrences to abstract upon. For example, in the cube of Figure 4a, if the two marked black regions are intensionally identified then structural abstraction can distinguish them from the other black region; this creates an abstraction that when applied to, say gray, yields the cube described in Figure 4b.

In music, structure abstraction allows for abstraction on particular note occurrences. For example, when working on the motif in Example 1 (from Section II), structure abstraction can be used to abstract only over the first and third occurrence of ‘ $\mathcal{E}_0$ ’, so that when applied to ‘ $\mathcal{G}_0 - \mathcal{E}_0$ ’ yields the motif in Example 3:

*Example 3:*

$$(\mathcal{C}_0, \frac{1}{4}) - (\mathcal{G}_0, \frac{1}{8}) - (\mathcal{E}_0, \frac{1}{8}) - (\mathcal{F}\sharp_0, \frac{1}{4}) -$$

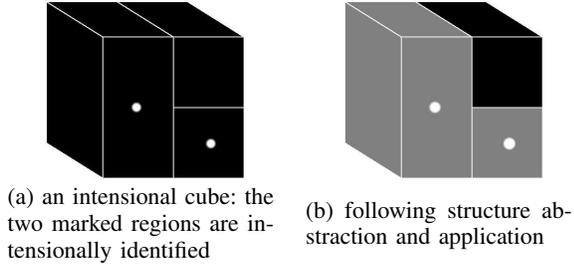


Fig. 4

USING AN INTENSIONAL CUBE

$$(\mathcal{E}_0, \frac{1}{8}) - (\mathcal{G}_0, \frac{1}{16}) - (\mathcal{E}_0, \frac{1}{16}) - (\mathcal{D}\#_0, 1)$$

#### D. A Formal Definition of Structure Abstraction

Concrete structure abstraction is meaningful if the expressions to which it is applied enable distinction and identification of occurrences of their components. That is, expressions can be conceived as single origin Directed Acyclic Graphs (DAGs), with leaves that correspond to atomic data values, and internal nodes that correspond to constructors (*i.e.*, constructors are applied to expression references). DAGs represent expressions:

- 1) A DAG that contains a single node represents the data value associated with this node.
- 2) A composite DAG with origin  $v$  and directed arcs to nodes  $v_1, \dots, v_n$ , represents the expression

$$\text{exp}[v] = \text{constructor}[v](\text{exp}[v_1], \dots, \text{exp}[v_n])$$

Note that the nodes  $v_1, \dots, v_n$  are not necessarily distinct. Two arcs that lead to the same node capture the intended identicalness of the arguments of the constructor of  $v$  (like the third person's hair and beard in Figure 3). Clearly, different DAGs can represent the same expression since references are lost in the above translation (recall the discussion on sameness and observability). The *meaning of a DAG* in a given domain is obtained by domain-specific evaluation of the expression represented by the DAG. Figure 5 shows two DAGs that represent the same expression. The expression evaluates to the cube in Figure 4a. Now we are in a position to introduce structure abstraction and application which are the two major constructors enabling concrete structure abstraction.

*Structure abstraction* is a two argument constructor (an internal node), that accepts a DAG — the *abstraction body*, and one of its nodes — the *abstraction node*. A DAG whose origin corresponds to the abstraction constructor is an *abstraction DAG* (see Figure 6). The nodes in the sub-DAG of the abstraction node (including itself) are called *bound* in the abstraction DAG. In an arbitrary DAG, nodes that are not bound are *free*. That is, given a DAG  $G$  and a node  $v$  in  $G$ ,  $v$  is bound if and only if  $v$  is an abstraction node, or a descendant of one, in an abstraction sub-DAG of  $G$ .

An abstraction DAG  $G$  is *valid* if its abstraction node is free in its body, and is the origin of a DAG  $G'$  whose nodes are not externally referenced. That is, except for the abstraction node, the nodes of  $G'$  are referenced only from

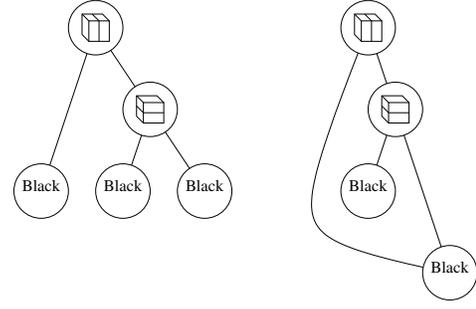


Fig. 5

TWO CUBE-DAGS THAT EVALUATE TO THE CUBE FROM FIGURE 4A

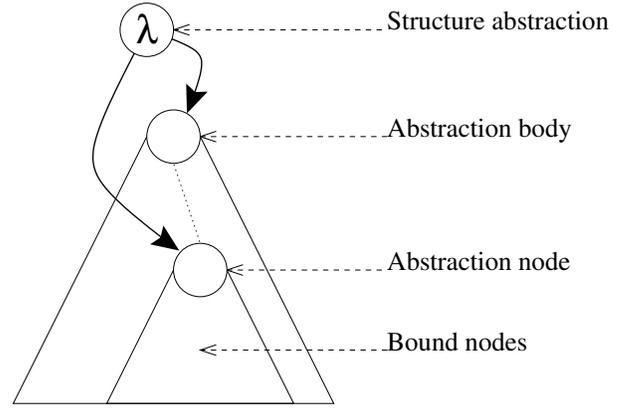


Fig. 6

AN ABSTRACTION DAG

nodes in  $G'$ . In particular, in nested valid abstractions, the outermost abstraction node cannot be within the sub-DAG of the inner abstraction. Figure 7 demonstrates a valid and invalid abstraction DAGs. Note that an abstraction DAG stands for an intensional abstraction, *i.e.*, a function on DAGs. Therefore, an abstraction DAG does not represent an expression (it is meaningless outside the DAG domain).

*Application* is also a two argument constructor intended to apply an abstraction on an argument. The arguments are labeled the *application operator* DAG, and the *application argument* DAG. A DAG whose origin corresponds to the application constructor is an *application DAG*. A *redex* DAG is an application DAG whose operator DAG is a valid abstraction DAG and is disjoint from the application argument DAG.

A reduction of a redex DAG is demonstrated in Figure 8. The reduction rule follows:

Let  $v, v_1, v_2$  be the origins of the redex DAG, its operator DAG, and its argument DAG, respectively. The reduction of this redex DAG is:

$$v \longrightarrow \text{body}(v_1)[v_2/\text{abs}(v_1)]$$

where  $\text{body}(v_1)$  stands for the origin of the abstraction body of  $v_1$ ,  $\text{abs}(v_1)$  for its abstraction node, and  $x[y/z]$  is the DAG obtained from  $x$  by replacing the

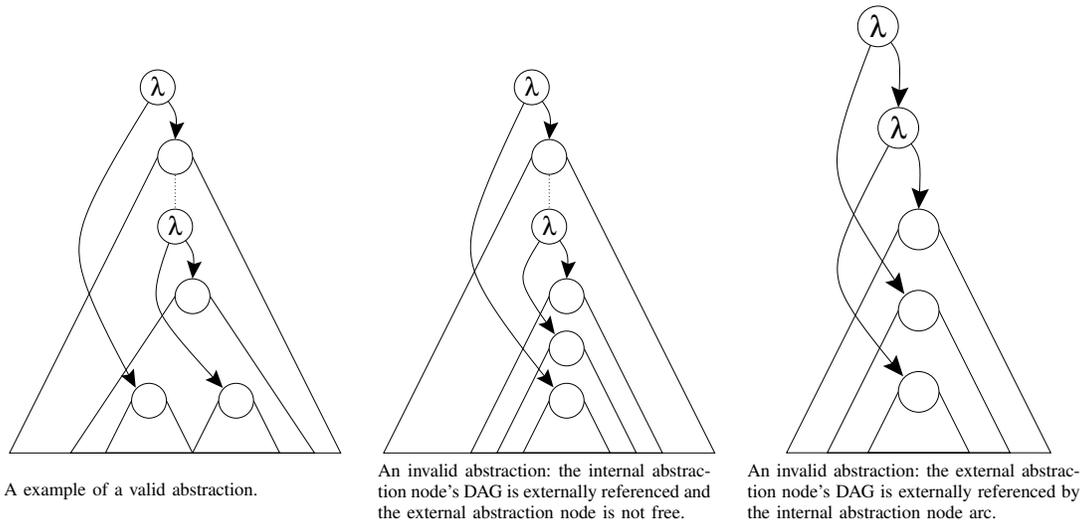


Fig. 7

ONE VALID AND TWO INVALID ABSTRACTION DAGS

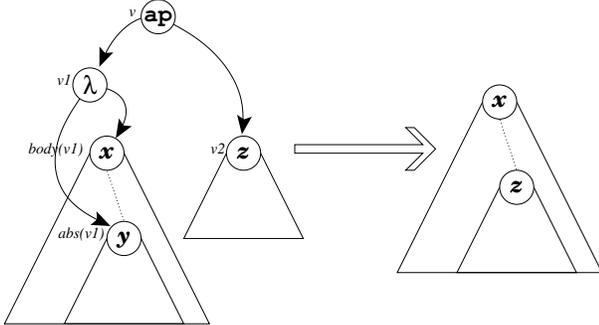


Fig. 8

REDUCING AN APPLICATION.

sub-DAG rooted in  $z$  by the one rooted in  $y$ <sup>10</sup>.

Note that since  $v_1$  is the origin of a valid abstraction DAG, its application does not leave “dangling references”, resulting from external references to nodes in the  $\text{abs}(v_1)$ . The intended DAG function of a valid abstraction DAG is implicitly defined by this reduction rule.

The structure abstraction and application rules introduce a DAG-based analogue of the Lambda calculus. It is well defined due to the above restrictions — valid abstraction DAGs and disjoint reduction arguments. Moreover, since the expressions are DAGs this formalization is simpler than Lambda Calculus. In particular, there are no problems with bound and free occurrences of symbols, conflicting substitutions, and different reduction orderings.

*DAG evaluation* is obtained by applying reductions until no more redex DAGs are available, then evaluating the expression that corresponds to the resulting DAG. Remember that in case the final redex-free DAG still contains abstraction or

<sup>10</sup>This presentation does not cope with replacement formally — one way to handle such side effects is to say that the whole DAG is copied, making the necessary modifications.

application nodes, then it does not represent an expression, so its evaluation is undefined.

The computational power of value and of structure abstractions is the same, since structure abstraction can simulate value abstraction, and structure abstraction can be simulated with *planned* lambda abstractions. The difference lies in the concrete abstraction mode. In the planned mode, the sharing and distinctions among multiple occurrences of a value are captured by sharing abstraction variables, or distinguishing among them, as demonstrated in the  $2 \times 2$  example in subsection III-B. However, in the concrete abstraction mode, value abstraction can generate only the  $\lambda x. x \times x$  abstraction, while structure abstraction can create both, since the sharing intension is kept in the DAG expression<sup>11</sup>.

### E. Structure Abstraction in an Interactive Environment

The formalization that was presented in Section III-D only handles a static mathematical world. When we get to a “real-world” interactive environment, we must consider side-effects in the form of substitutions that *change* the DAG. This is dangerous since abstractions that were made can change their functionality or even become invalid due to changes.

This means that a slightly different strategy should be employed to cope with these changes: when an abstraction is made, its DAG is copied to an abstraction environment. This environment is an association list that binds names with stored abstractions; it forms a read-only memory for these abstractions which means that they cannot change. If a valid abstraction is made, referring to it using its name always returns a copy which is the same as the original one<sup>12</sup>.

<sup>11</sup>An idea that was suggested by Orlarey is the concept of *generalized abstractions*, where a simple abstraction over some construction can take any value as an argument, and application will take the form of pattern-matching and substitution. The structure abstractions presented here can be viewed as a much simplified form of generalized abstraction.

<sup>12</sup>Note that this does not contradict a mechanism to delete bindings or redefining them — the only guarantee we need is that stored DAGs never change.

Using this approach, abstractions are not used directly in DAGs: a new kind of node is introduced that is a name reference. The application reduction rule is changed accordingly, a redex has a valid name reference as its operator, and reduction is performed using the abstraction DAG referenced by this name. This is similar to the way *BOOMS* is implemented<sup>13</sup>, see Appendix I for more details.

#### IV. COMBINING THE TWO ABSTRACTIONS IN A SINGLE TOOL

In the previous section we have seen that concrete structure abstraction is more powerful than concrete value abstraction since it uses identities to express sharing of subcomponents in an intuitive way. In this section we argue that, nevertheless, it cannot be used as a replacement for concrete value abstraction. This seemingly contradictory argument results from the truly contradictory nature of concrete abstraction and a fully structured intensional object. Concrete abstraction is intended to support a creative spontaneous mode of work, where abstractions arise in an associative manner, out of concrete (extensional) objects, *e.g.*, a cube, a music piece, or a document. Manipulation of a structured intensional DAG object, on the other hand, must be planned, since the complex structure usually clobbers the value of the expression to which it evaluates. Hence, it is hard to come up with associatively created abstractions.

Consider, for example, the DAG in Figure 9 that represents the intensional cube from Figure 4a. The actual cube intended by this DAG expression is quite obscure. Consequently, although the intensional DAG structure identifies the first and third occurrences of black and enables concrete abstraction on them, it is unlikely that a user will come up with that idea, unless it was planned in advance — but in that case concrete abstraction is not necessary in the first place. We see that the straightforward generalization of elements in concrete objects, which is the essence of concrete abstraction, is not possible if the user interacts only with the structured object.

We claim that both forms of concrete abstraction are necessary for supporting a creative, associative mode of object development. Concrete value abstraction is the natural mode of operation, while the intensional structure of the manipulated objects arises from the history of object development. If the management environment keeps both the concrete object and the history of its development, then abstractions can emerge in a natural way from the concrete object, but be reflected and performed in its associated intensional object. For example, Figure 10 presents a dual simultaneous view of the cube from 4a, and its intensional DAG structure from Figure 9. The DAG might have been developed during the construction of the cube. If users can simultaneously observe the cube and its DAG, then they might realize that the first and third occurrences of black are based on a single object, and that an abstraction on this object is desirable. Similarly, in music, a composer wishes, typically, to work with the actual music piece, in a bottom-up mode. Using a structure-enabled tool, the intended structure of

<sup>13</sup>Such a store is also useful for named objects that allow *named copy-paste operations*.

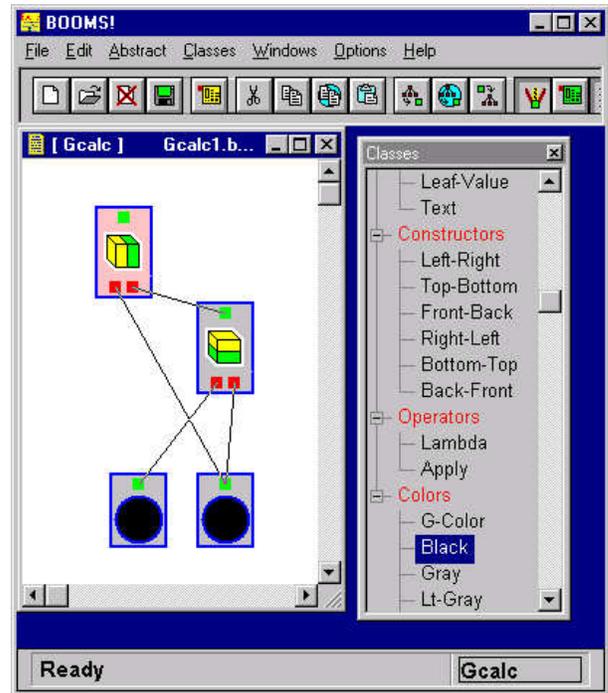


Fig. 9

THE STRUCTURE OF THE CUBE FROM FIGURE 4A

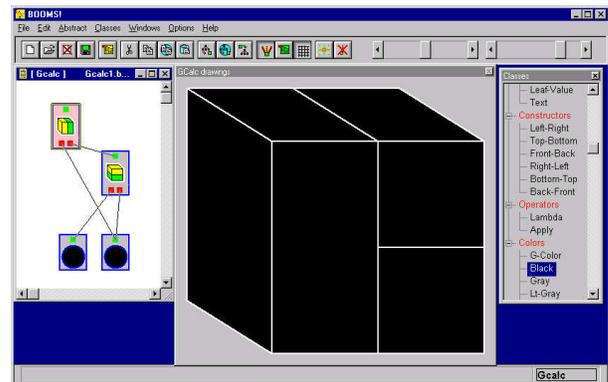


Fig. 10

A DUAL EXTENSIONAL-STRUCTURAL VIEW OF THE CUBE FROM FIGURE 4A

the composed piece can be maintained together with the piece, and structure abstraction can be applied to the DAG structure, by observing the actual piece and its structure simultaneously. The main point is that this abstraction is possible and feasible only if both the extensional and the intensional objects are equally accessible to the user.

The above discussion leads to the conclusion that a powerful concrete abstraction requires a *double-view historical* editing tool, that provides smooth integration of the extensional and intensional views of an object in a single end-user tool. Such an interface should give the “look and feel” of an extensional editor, keeping a structural representation that is held

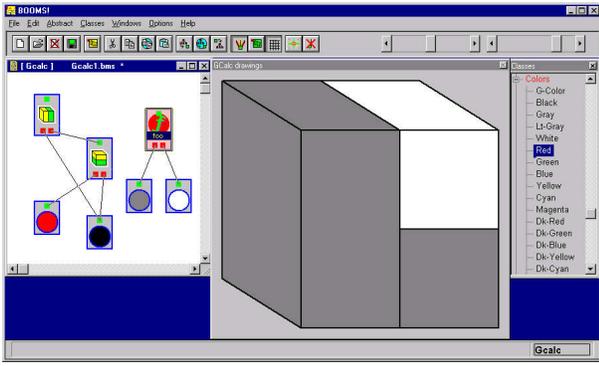


Fig. 11  
STRUCTURE EDITING IN *BOOMS*.

internally. The tool should enable the user to switch, easily, to a structure editing mode, expressing intentions. Moreover, we claim that the intensional structure of the object being developed can emerge from the history of user operations during object development. That is, certain operations can be viewed as having indication for intentions. For example, a copy-paste operation usually means that the newly created subcomponent copy is actually identical to the first in an intensional sense. *BOOMS* is a prototype tool that supports double-view editing. Figure 11 demonstrates a *BOOMS* session for editing the intensional structure of the cube from Figure 4a: first, a structure abstraction was made from the given DAG and named “foo”, then a red cube replaced the first black cube and finally an instance of this abstraction was created and given white and gray colors as arguments. In the next section, we describe a model for a double-view editor that integrates the extensional and intensional views of objects, and enables smooth interactions between the two modes.

Still another benefit of a double-view development tool has a flavor pf educating users. The vision is that such a tool allows beginners to work in a purely extensional environment, and switch to the structured view from time to time for exploring the structure that is being created. In later stages, users learn the *meaning* of the structure and its usefulness — they casually start editing the structure to better represent their intended structure. Eventually, they fully exploit the two views, switching as needed.

## V. AUGMENTING END-USER PROGRAMMING IN EDITORS WITH CONCRETE ABSTRACTION

End user programming aims at giving the user the option for doing a complex sequence of operations once, then abstracting it into a “reusable operation” that can be applied later on in different contexts. It can be viewed, in itself, as an editing domain whose values are stored in buffers, and are created by constructors that are primitive edit operations like **create**, **compose**, **delete**, **select**, **copy**, **change attribute**. The exact nature of values vary, of course. In Notepad a value is a sequence of characters, in Paintbrush a value is a bitmap, in *Word* a value is a sequence of characters and objects with

attributes. Abstraction in this domain yields templates of the domain values.

The main difficulty with which end-user programming has to cope is that a “standard” naive user is primarily interested in getting some document done, and is not willing to put efforts into improving the editing process. Therefore, end-user tools should have an associative, immediate flavor, that does not require planning on the user’s part. Concrete abstraction seems like a natural mechanism since it emerges in an associative way within the editing session, rather than being a planned in advance activity.

The kinds of possible concrete abstractions in an editing domain depend on the nature of the values being stored. The richness of the editor buffer has direct impact on the power of the concrete abstractions that it can support. We distinguish four levels of value domains:

**Extensional values:** Domain values are fully computed at each primitive editing operation. Such editors keep no structure at all, for example, PaintBrush.

**Intensional structured values:** Domain values keep the intended structure, as in the cubes or the music domains, but have no commitment to the structure as entered during an editing session. For example, a normalization procedure can be used to optimize the structured values and reduce them into some observable based canonical form.

**Intensional historically structured values:** Domain values keep the intended structure, as entered during an editing session — no information on cloning and destructive operations is kept. The GCalc editor as well as many modern editors like XFig, MacDraw and *Coreldraw* keep buffered values of this kind.

**Intensional identity-based historically structured values:** Domain values are DAGs, called *edit graphs*, that keep the intended structure, as entered during an editing session. *BOOMS* keeps values of this kind, Windows’ OLE provides a similar but restricted capability for applications — embedded “links” to objects allow users to specify sharing of the same object.

The kind of values that an editor stores is a design decision taken by the implementation designer. Extensional values are probably the easiest to manipulate, but support weak editing operations: An **undo** operation requires storing snapshots of buffer states; a **redo** operation is not possible. Intensional, normalized structured values may be optimal, free of redundancies, and enable comparison of values that were built in different ways. However, such values can support concrete value abstraction on the normalized structure alone, and not over the sequence of user operations. Intensional, historically sensitive, structured values can support concrete value abstraction on user operations, as demonstrated in GCalc. Still, the structured values are pure values, and keep no identities for their components. In particular, they do not keep track of cloning and of destructive operations. A **delete** operation is fully computed and removed from the editing history.

Only the DAG-based values can store the full scale of editing operations. An edit graph can support structure sharing among its components, as demonstrated in the introduction of concrete structure abstraction. This way intensional dependen-

cies among objects that imply that changing the properties of one object can change its occurrences in other contexts can be guaranteed. For example, a **copy** operation implies structure sharing between the source and the target objects. Similarly, a **grouping** operation implies structure sharing between the objects being grouped to the group object itself. The following example emphasizes the value of structure sharing:

```
(define A (+ (+ 1 2) 3))
(define B (+ 1 2))
(define C (+ B 3))
```

Although ‘A’ and ‘C’ evaluate to the same number, an intentional, identity-based, historically structured editor should not optimize ‘C’ to be internally equal with ‘A’ because it then loses the structure sharing with ‘B’.

Since DAG values can support concrete structure abstraction, they enable full concrete structure abstraction of user operations. This is important, especially for supporting editor operations like **undo**, **redo**. If the editor keeps a fully historical edit graph then concrete structure abstraction can be used to define **redo** abstractions. That is, take a sequence of edit operations like **group**, **copy**, **paste**, apply abstraction to any of its components, and then apply the abstracted sequence in a different context, to different components.

We suggest to use the double view approach described in Section IV to implement a historical editor. The edit graph can evolve internally, as a result of user operations. The user can interact with a regular extensional editor buffer, but can consult the structured view for more complex editing operations. **redo abstractions** can be defined as an end-user structure abstraction applied to the edit graph. These capabilities are partly implemented in *BOOMS*, which is described in Appendix I.

## VI. RELATED WORK

### A. Computer Music Environments

**Common Music** : *BOOMS* is inspired from music composition environments and from graphical and historical editors in general. In composition environments, the need to support creativity requires the user to be endowed with programming capabilities. *Common Music* [8], for example, is a powerful and popular environment for algorithmic composition. It provides a rich set of music primitives, types, and operations, including advanced notions such as music streams, and hierarchy supporting containers. *Common Music* is embedded in Common Lisp, and the user can program the composition using the notions supported by the system. A professional usage of *Common Music* requires full programming skills.

**DMIX** : A different approach for supporting creativity in music composition is employed in real time composition environments like *DMIX* [5] and *Elody* [4], that replace (or extend) planned programming with direct visual composition. *DMIX* [5] has an expressive user-interface that 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 yields an interesting form of abstraction in the following way. One 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.

**Elody** : The *Elody* environment of Orlarey [4] supports true concrete abstraction, as discussed in detail earlier in this paper. *Elody* can be viewed as a visual functional language that is grounded in the music domain. Furthermore, following the pure functional tradition, *Elody* does not distinguish data from functions. Consequently, compositional processes can be applied to high-order functions, so to yield high-order scores, and music objects can be considered as functions as well. According to its designers, *Elody* can be viewed as an active music notation, since its programs are also scores.

### B. Historical and Structural Editing

**Chimera** : The double-view historical editor that we propose is influenced, to a great extent, by the *Chimera* graphical editor of Kurlander [9]. The aim of *Chimera* was to investigate ways to automate repetitive tasks in user interfaces. *Chimera* lets the user “program an application through its user interface.” Five powerful techniques were developed to automate repetitions; the most relevant to this work are *editable graphical histories* and *macros by example*.

Graphical histories encode in a “comics 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” [9, p.11]. The graphical history is automatically maintained as the user performs actions on the editor, and 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). The main difference is that *Chimera* depicts histories in the same visual language as the object editor, while *BOOMS*, motivated by the need to denote structure in music composition, depicts histories in a hierarchical view, focusing on their structural properties. That is, *Chimera* depicts histories

as sequences of “cartoon-like” pictures. Each picture shows a sequence of operations as a snapshot of the editor with graphical annotations that indicate where the modification happened. In contrast, in *BOOMS*, histories appear in a hierarchical view. Each operation is depicted by an iconic node in a tree with arcs pointing to the parameters of the operation. The visual language used for histories is, therefore, quite different from the one used in the editor itself. It is a graphical rendering of the procedures applied during the editing session. This depiction highlights more clearly the structural properties of the resulting objects: for example, it shows when an element in the editor is shared by several operations (the same node is reached by several arcs). It provides an easier basis for the type of abstraction we are advocating, but it is less readable than the Chimera approach. We believe that a combination of the two visual languages can offer the “best of both worlds.”

**The Programmer’s Apprentice** : Another historical editor that is somewhat relevant is the *Programmer’s Apprentice* project [10] whose objectives were 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 was encoded in the form of *clichés*, which encodes the knowledge shared by the programmer and an external assistant when modifying a piece of code. 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.

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. The *Programmer’s Apprentice* project illustrates the need to maintain information beyond the edited extensional object, in order to describe the intention of the designer.

In *KBEmacs*, the integration of the domain value editing and knowledge editing is through a stage of automatic analysis (plan recognition) of user actions. While this approach is conceivable in the programming domain, where a large cliché library can be designed, it is much harder to apply in the music domain (or any other creative 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.

### C. Double-View Editing

Double-view editing is important for any editor that stores and operates on non-extensional values. Therefore, the extension of  $\LaTeX$  tools with the “Xdvi” tool, which extends the editing session with an extensional view, was a major improvement to  $\LaTeX$  document editing. Yet, in the  $\LaTeX$ -Xdvi combination only the  $\LaTeX$  view is editable.

**Lilac**: [11] is a true double-view document editor. It consists of a *page view* which is a “What-You-See-Is-What-You-Get” editor and a *source view* which is an intensional editor, that describes the document as a program written in the Lilac document language. This language enables the user to define new document structures. Both views are maintained by hierarchical data structures: The source view by a *syntax tree* which represents the parsing of the document, and the page view by a *display list* which represents the hierarchical geometrical relationships of the syntactical components. The editor supports two way mapping relationships between the page view image to the display list, between the display list to the syntax tree and between the syntax tree to the source view. Therefore, both views are editable.

The author admits that 95% of the time he spent on the page view alone, while the source view is used mainly for editing complicated structures, for global styling, or for creating new styles and constructs. The *BOOMS* approach is clearly influenced from Lilac — the DAG object of *BOOMS* corresponds to Lilac’s hierarchically maintained source view; the three different applications correspond to three different page view applications. In that sense, *BOOMS* generalizes Lilac into a domain independent double-view editor, and uses the two forms of concrete abstraction as a main programmability means. However, *BOOMS* is different from Lilac in two major aspects:

- the structure view is a graphic icon-based structure editor rather than Lilac’s source text view,
- There is no relation to abstraction in Lilac.

## VII. CONCLUSION

In this paper we propose to use concrete abstraction as a unifying tool for extending editors with end-user programmability, in a domain independent way. It is appropriate for creative domains where abstractions are not planned but emerge from concrete examples, and for helping users to deal with repetitive tasks or define new primitive operations. We have shown that concrete abstraction can be used for creating templates of structured objects in the editor subject domain,

and of sequences of user actions and behaviors. We argue that concrete structure abstraction that is based on an edit graph is particularly suitable for historical editor programmability that includes **undo** and **redo** operations.

Our approach combines the concrete abstraction of GCalc that enables powerful end-user computing with the historical editing of Chimera and the double-view editing mode of Lilac. It extends the concrete value abstraction of GCalc with concrete structure abstraction, it extends the historical view of Chimera with hierarchy and structure sharing, and it extends the double-view architecture of Lilac to general domains, not necessarily text documents.

*BOOMS* is a working initial prototype for a double-view historical editor that features concrete abstraction as a major end-user computing means. The generality of *BOOMS* is demonstrated by three different applications in the music, graphics and symbolic arithmetic domains.

Future work in this direction is required. On the theoretical level, concrete structure abstraction can be further studied and possibly extended, to allow for abstractions that are currently restricted. On the empirical level the integration of the two views in *BOOMS* should be further developed. In addition, the history edit graph can be further studied, and simplification algorithms and a normal form should be developed.

## APPENDIX I *BOOMS*

The *BOOMS* project, described in [2], 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. Structure is, therefore, a major motivation behind *BOOMS*, following [12]. Existing music editors do not provide appropriate support for composition: most only address the score editing process. Given this state of affair, composers must add personal annotations to their compositions to remember their structure.

*BOOMS* presents an editor intended for music composition. It 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 turning 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 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 GCalc.

In a *BOOMS* instantiated editor, the end-user interacts with a hierarchical editor that manages editing histories. The final goal is to have a full double-view editor that will allow standard editing operation while, at the same time, the hierarchical view is maintained. The editing graph can be edited by the user to specify structural intentions. Abstraction can also be performed on the editing graph, turning a sequence of editing operations into a new reusable function.

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.

## REFERENCES

- [1] 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.
- [2] Eli Barzilay, "Booms: Booms object oriented music system", Master's thesis, Ben-Gurion University, Computer Science, 1997.
- [3] Y. Orlarey, D. Fober, and S. Letz, "Elody: A java + midshare music composition environment", in *International Computer Music Conference*, 1997.
- [4] Y. Orlarey, D. Fober, and S. Letz, "The role of lambda-abstraction in elody", in *International Computer Music Conference*, 1998.
- [5] D.V. Oppenheim, "Dmix: A multi faceted environment for composition and performing computer music: Its design, philosophy, and implementation", in *Computer Music Days*, Delphi, Greece, 1992, pp. 1–10.
- [6] A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, 1941.
- [7] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1986.
- [8] H. Taube, "Stella: Persistent score representation and score editing in common music", *Computer Music Journal*, vol. 17, no. 4, pp. 38–50, 1993.
- [9] D.J. Kurlander, *Graphical Histories*, PhD thesis, Columbia University, Computer Science, 1992.
- [10] Charles Rich and Richard C. Waters, *The Programmer's Apprentice*, Addison-Wesley Publishing Company, 1990.
- [11] K.P. Brooks, *A Two-View Document Editor with User-Definable Document Structure*, PhD thesis, Stanford University, Computer Science, 1988.
- [12] M. Balaban, "The music structures approach to knowledge representation for music processing", *Computer Music Journal*, vol. 20, no. 2, pp. 96–111, 1996.



**Mira Balaban** holds a Ph.D. in computer science from the Weizmann Institute of Science, and a music degree from the Rubin Academy of Music in Tel-Aviv. She is now affiliated with the Department of Information Systems Engineering in Ben-Gurion University ([mira@cs.bgu.ac.il](mailto:mira@cs.bgu.ac.il)). Her research is in the areas of knowledge representation, conceptual modeling, database semantics, programming languages, and computer music.



**Eli Barzilay** is a Ph.D. candidate in the Computer Science department in Cornell University. His research interests include applied logic, formal systems, programming languages, and computer music. He received a B.Sc. and an M.Sc. in computer science from Ben-Gurion University in Israel. He can be reached at [eli@barzilay.org](mailto:eli@barzilay.org).



**Michael Elhadad** is a Senior Lecturer at Ben Gurion University. His research interests include intelligent user interfaces and computational linguistics. He received a Ph.D. in computer science from Columbia University. Contact him at the Dept of Computer Science, Ben Gurion University, Beer Sheva 84105, Israel, [elhadad@cs.bgu.ac.il](mailto:elhadad@cs.bgu.ac.il).