Journal of Visual Languages and Computing (1996) **7**, 219–242

Programming in Three Dimensions

Marc A. Najork

Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, U.S.A. najork@src.dec.com

Received 4 May 1995; revised 7 December 1995, 26 January 1996; accepted 26 January 1996

Cube is a three-dimensional, visual, statically typed, higher-order logic programming language, designed to be used in a virtual-reality-based programming environment. In this paper, we give an informal overview of the language and describe a prototype implementation.

1. Introduction

Programming is the activity of describing an algorithm in a formal notation—a programming language—for the purpose of executing it on a computer. The vast majority of programming languages are textual in nature; that is, programs are described by an essentially one-dimensional string of characters.

Visual programming languages, in contrast, use a predominantly graphical notation to encode an algorithm. The central argument in favor of visual languages is that humans are known to process pictures easier and faster than text. According to Raeder, 'it is commonly acknowledged that people acquire information at a significantly higher rate by discovering graphical relationships in complex pictures than by reading text' [29].

One of the reasons for this phenomenon is that the human sensory system is set up for real-time image processing. Two- or three-dimensional pictures can be accessed and decoded more rapidly than linear text. So, using a visual notation allows us to shift part of the user's cognitive load from the conscious level to the visual cortex. A related aspect is that text is sequential in nature, while pictorial representations provide random access to any part of the picture, as well as detailed and overall views.

A second important reason is that visual representations provide a syntactically rich language. Text is essentially one-dimensional, while visual representations can employ two or three spatial dimensions as well as other properties such as color.

The first visual programming system was implemented in 1966 by William Sutherland [38]. Since then, a multitude of visual languages have been developed. Overviews of the field can be found in [3] and [11].

This paper describes Cube [23, 25], the first visual language to employ a three-dimensional (3D) notation. Apart from its novel syntax, Cube is innovative in other aspects: it is the first visual language to apply the dataflow paradigm to logic programming; it is among the first visual languages with a static polymorphic type system, and guarantees the absence of run-time type errors; and it is a higher-order language, meaning that it treats predicates as first-class values. The main part of the

1045-926X/96/020219 + 24 $18.00/0

paper elaborates on these innovations. At this point, however, we would like to motivate Cube's most obvious innovation, the use of a 3D syntax.

The possibility for using a 3D notation has been realized for quite a while. In his 1987 paper 'Out of Flatland: Towards 3-D Visual Programming', Ephraim Glinert argued that the time is ripe for exploring the possibilities of 3D notations: 'But first, why do we advocate programming in three dimensions? . . . It is time for computer science to begin exploring revolutionary rather than evolutionary means of programming, in the hope that the tools will be ready when required.' [10]

We have identified four potential benefits of using a 3D notation: it can alleviate the screen space problem; it allows for better graph layout; it can convey additional semantic information; and it can facilitate new interaction environments. Let's look at these four points in more detail:

- One common criticism against visual languages is that they use the precious screen real estate less efficiently than textual languages. This phenomenon is commonly called the *screen space problem.* It is not particular to visual languages but also known in other domains such as visualization. In many of these domains, 3D has been used to alleviate the screen space problem. A good example is the Xerox PARC Information Visualizer [32], which uses a multitude of 3D representations of abstract structures (trees, tables, etc) to make more effective use of available screen space. Using 3D provides us with a larger design space: the additional depth dimension allows us to squeeze more information into the same screen real estate, and interactive rotation of the scene allows us to retrieve different parts of this information. To quote Robertson *et al.,* 'it seems plausible (but not yet proven) that 3D can be used to maximize effective use of screen space' [32].
- A second advantage of a 3D notation is that it is easier to lay out graph structures in 3D than in 2D (since it is always possible to avoid overlapping arcs in 3D), and that graph structures are more easily comprehended when being displayed in 3D. This is particularly relevant to visual languages because a large fraction of them (including Cube) use a dataflow notation, i.e. are essentially represented as graphs. Ware and Franck have performed an empirical study on user comprehension of graph structures and found that users can extract three times as much information from a graph that is displayed in 3D (using a stereo display and head coupled perspective) than from a graph that is displayed in 2D [40].
- In his seminal survey of graphical programming techniques [29], Raeder argues that text is an essentially one-dimensional stream of words. Pictures, on the other hand, provide several spatial dimensions along which to lay out information, together with a host of other properties, such as color, texture, shape, etc. Visual languages are therefore syntactically 'richer' than textual languages. By this argument, 3D languages are richer than 2D languages. The extra spatial dimension that 3D offers also provides us with an extra dimension of expression. Of course, this is true only if the third dimension is actually used to encode additional information. (As we shall see, Cube uses dimensional extent to express logical operations, e.g. conjunction and disjunction.)
- A final potential reason for visual languages to adopt a 3D notation is that such a notation naturally complements a Virtual Reality (VR) environment. Virtual

Realities have potential as comfortable programming environments due to their very direct and intuitive mode of interaction. Instead of moving a mouse in order to manipulate an object in the scene, the user can manipulate the object directly. VR environments also promise to alleviate the screen space problem even further, by coupling the viewpoint and perspective to the user's head position and orientation and thereby providing tight and intuitive control over what part of the design space is visible. Another reason why one might want to program in a VR environment is for developing VR software. This was the motivation for the work on Lingua Graphica [37], a 3D visual language developed at Lockheed's AI Center. The key idea is that developing VR software within a VR programming environment shortens the edit–compile–debug cycle.

The remainder of the paper is structured as follows: First, we give an informal overview of Cube, based on a number of examples. (For a formal definition, the reader is referred to [25].) Then, we describe a prototype implementation of a Cube interpreter and programming environment. We review some of our language design choices, compare Cube to related work and finally offer some concluding remarks.

2. The Cube Language

Semantically, Cube is a logic programming language, similar to Prolog [5]. Prolog, however, is a first-order language, while Cube is higher-order, meaning that predicates are first-class values and can be passed as arguments to other predicates. The expressive power of higher-order languages is well-known. Higher-order features are standard in functional languages and have been incorporated into a few (textual) logic languages as well, such as λProlog [21] and HiLog [4].

A second difference between Prolog and Cube is that Prolog is an untyped language, whereas Cube has a static polymorphic type system. The Cube system uses a variant of the Hindley-Milner type inference algorithm [7] to ensure that programs are well-typed; such well-typed programs are guaranteed to never encounter a run-time type error.

The remainder of this section describes Cube informally, based on a number of graduated examples.

2.1. The Dataflow Metaphor

Consider the simple program shown in Figure 1. It consists of two transparent cubes that are connected by a *pipe.* The transparent cubes are called *holder cubes*; they may contain terms (which are represented by cubes as well), and thus correspond quite closely to variables in a textual language.

The left holder cube contains a term: an opaque cube with the icon '1' on its top side. This cube is called an *integer cube,* and represents the integer 1. The two holder cubes are connected by a *pipe* which serves as a 'conduit' for values. The metaphor we use here is the dataflow metaphor: during evaluation, a value contained in a holder cube flows to all the other holder cubes connected to it. If a holder cube receiving a value is empty, it will be filled with this value, if it already contains a value, the two values must be *unifiable*; both holder cubes will then contain the same value, namely, the most general unifier of the two values. If this is not possible, the dataflow *fails.*

Pipes have no particular directionality: data can flow through them in either direction, in fact it can flow in both directions at once. It should also be noted that the value contained in a holder cube never gets changed but only refined.

We can extend the analogy we have drawn between Cube and textual languages: Holder cubes correspond to *logic variables,* and connecting two holder cubes by a pipe corresponds to unifying two logic variables. Furthermore, a holder cube containing a term cube (such as an integer cube) corresponds to a logic variable unified with a term.

In the textual framework, a unification is a special case of an atomic formula. A Cube program (i.e. the entire 'virtual space' in which Cube expressions are located) corresponds to a query in a textual logic language, that is, a conjunction of all the atomic formulas.

Figure 2 shows the Cube program of Figure 1 after evaluation. The integer cube 1 has flown from the left to the right holder cube (intuitive interpretation); or the left holder cube was unified with 1 and with the right one, leaving both instantiated to 1 (logic interpretation).

Some Cube queries do not have any solutions. For example, if the right holder cube in Figure 1 contained a value other than 1, say 2, the dataflow between the two holder cubes would fail because 1 and 2 are not unifiable, thereby causing the entire computation to fail.

2.2. A First Glimpse at Types

We have mentioned before that Cube is a statically typed language and uses a type inference system. The user can trigger the type inference; the Cube system will then determine if the program is well-typed and, if so, will indicate the type of every empty holder cube by placing a *type cube* inside it. A type cube is an opaque grey cube with an icon on its top that identifies the type. There are three predefined base types: integers, floating-point numbers and propositions. Cube also allows the user to define new types, such as characters, strings, lists, or trees.

Given the program from Figure 1, Cube will infer that 1 is an integer, so the left holder cube contains an integer, and therefore the right holder cube must contain an integer as well. It will thus fill the right holder cube with a type cube with the icon 'Z' on its top, representing the integer type (see Figure 3).

What happens if Cube cannot determine the type of a holder cube? In other words, how do we represent uninstantiated type variables? An uninstantiated type variable is shown as a grey type cube with a unique index in the northwest corner of its top side. The same concept is used for uninstantiated variables: a variable of type τ is shown as τ's type cube with a unique index in the southeast corner of its top side; the cube is green because it represents a value. (Figure 17, appearing in a subsequent section, gives an example of uninstantiated variables and uninstantiated type variables.)

2.3. Predicate Applications

A *predicate cube* is represented by an opaque green cube with an icon on its top that identifies the predicate. A predicate cube typically also has a number of 'holes' in its sides: cubic intrusions with a transparent cover on the outside and an icon on top of

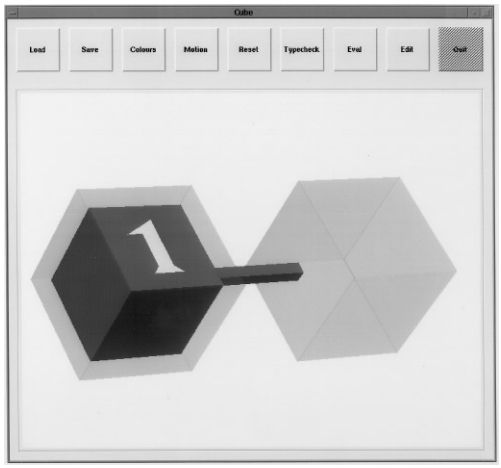

Figure 1. Value 1 flowing into empty holder cube

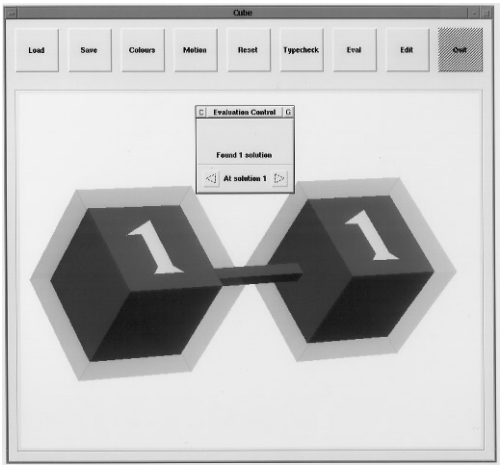

Figure 2. Value 1 has flowed into empty holder cube

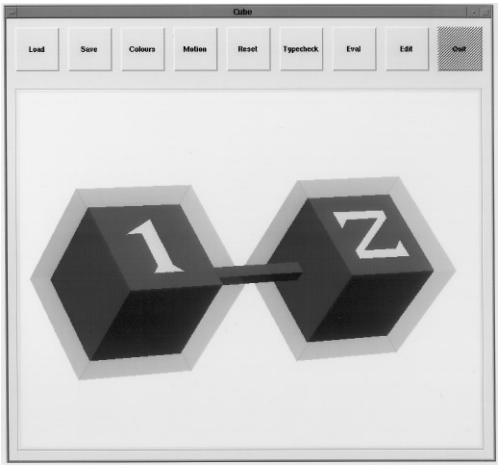

Figure 3. Program from Figure 1 after type inference

it. The 'holes' are called *ports* and serve as arguments to the predicate. They may be moved around freely over all six sides of the predicate cube; thus, an icon is needed to identify each port. A port is a special case of a holder cube (hence the transparent cover), and as such it can be connected to pipes and can be filled with a value.

So how can we apply predicate cubes? Assume we want to build a program to convert temperatures between the Celsius and Fahrenheit scale, which are related by the formula $F = 1{\cdot}8C + 32$. The program shown in Figure 4 is the Cube representation of this relation. It consists of two empty holder cubes (corresponding to C and F) and two holder cubes filled with floating-point values 1·8 and 32·0, respectively. It also contains a predicate cube referring to the floating-point multiplication predicate, and another predicate cube referring to the floating-point addition predicate. Both

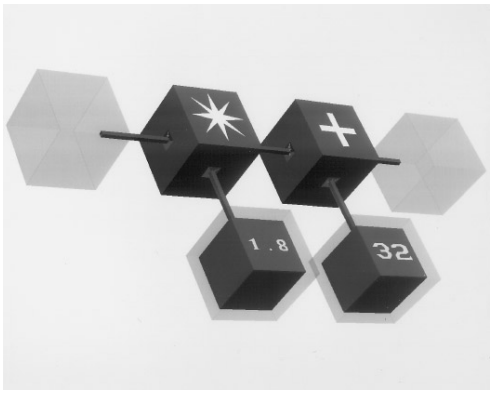

Figure 4. Temperature conversion

predicate cubes have three ports each, since in a logic programming context, addition and multiplication are viewed as ternary predicates. The first port of the multiplication predicate is connected by a pipe to the leftmost empty holder cube, the second one is connected to the holder cube containing the value 1·8, and the third one (the 'result') is connected to the first port of the addition predicate.

Thus, if the user puts a value into the leftmost holder cube, the multiplication predicate will receive this value, will multiply it by 1·8, and transfer the result to the addition predicate, which then adds 32·0 to it, and transfers the result to the rightmost holder cube.

Alternatively, if the user puts a value into the rightmost holder cube, it will flow into the 'result' port of the addition predicate cube, which will now *subtract* 32·0 from it, and transfer the result of this subtraction to the 'result' port of the multiplication predicate cube. This cube will *divide* the result of the subtraction by 1·8, and transfer the result of this division to the left holder cube.

This example demonstrates that arithmetic predicates work in either direction. Addition, for instance, can use the first two arguments to produce the third one, or the last two to produce the first one. The 'multidirectionality' of predicate applications complements the bidirectionality of dataflow in Cube.

In general, the addition predicate needs to know at least two values to determine the third one; otherwise, evaluation of the predicate is suspended until sufficient information is available. There are Cube programs which cannot be 'solved' because not enough information is available. The temperature conversion program, with neither a Celsius- nor a Fahrenheit-value supplied to it, is such a program. If we try to evaluate it, the system reports that it is unable to decide whether the query is satisfiable or not.

2.4. Predicate Definitions

Cube also allows us to define new predicates, thereby providing a mechanism for 'procedural abstraction'. Moreover, these predicates may be recursive (that is, they may refer to themselves), which allows us to describe potentially unbounded computations.

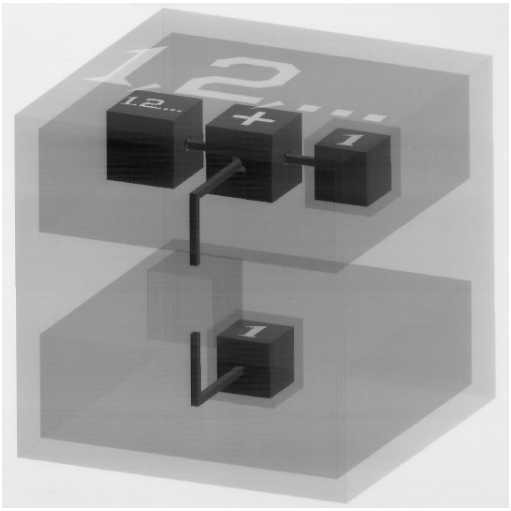

Figure 5. A natural number generator

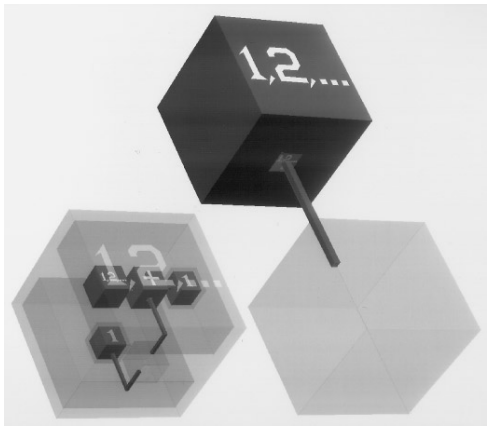

Figure 6. Program computing all natural numbers

Assume we want to define a predicate that generates all the natural numbers, i.e. the integers greater than or equal to 1. The natural numbers can be recursively defined as follows: (a) 1 is a natural number, and (b) if n is a natural number, then so is $n + 1$.

Figure 5 shows the Cube definition of this predicate. The outer cube, called a *predicate definition cube,* is a transparent green cube with an icon on its top. This icon provides a name for the new predicate. The small transparent cube set into the center of the front side of the definition cube is a *port*; it represents the formal parameter of the predicate under definition. The icon on its outer side identifies the port.

Inside the predicate definition cube are transparent boxes called *planes.* Planes are stacked vertically; in Cube, vertical arrangement (in the value world) indicates disjunction, while horizontal arrangement indicates conjunction. To draw the analogy to Prolog, each plane corresponds to a clause of a Prolog program.

Predicate definition cubes and planes may contain local predicate definition cubes. Predicate definitions occurring at the top level are visible to the entire program (including each other), predicate definitions local to another predicate definition cube are visible to all objects inside this cube, and predicate definitions local to a plane are visible to all objects inside the plane.

The lower plane forms the base case of the recursive definition. It contains a holder cube, filled with the value 1, which is connected by a pipe to the port representing the formal parameter.

The upper plane forms the recursive case of the definition. It contains an addition predicate cube whose first argument is connected to a recursive application of the natural-number predicate, while the second port is connected to a holder cube containing the value 1, and the third argument is connected to the port representing the formal parameter. Note that the recursive case of the natural-number predicate is represented by an opaque cube, i.e. a predicate cube. The icon on its top indicates which definition cube it refers to—in this case, the surrounding definition cube. The port in its side carries the same icon as the port of the surrounding definition cube;

for predicate cubes with several parameters, these icons are used to match up actual with formal parameters. Since predicate cubes refer to predicate definitions, they are also known as *reference cubes.*

The intuitive meaning of a predicate reference cube is that we would replace it by its corresponding definition cube (after moving the ports around to match them up). This intuition corresponds exactly to what is known as call-by-name semantics in textual programming languages.

If we pose a query like the one shown in Figure 6, we can imagine that the large reference cube referring to the natural-number predicate gets replaced by ('expanded to') the corresponding predicate definition cube. (In the actual implementation, the reference cube remains opaque.) The value 1 then flows from the holder cube in the lower plane of the expanded reference cube through the pipe into the port and from there out of the expanded reference cube and into the large empty holder cube. This constitutes the first solution to our query.

We can also imagine that not only the large reference cube was replaced by the definition cube, but that at the same time the recursive reference cube inside the top plane of the expanded reference cube was replaced by the definition cube as well (and the recursive reference cube inside *this* cube as well, and so on *ad infinitum*). So, the value 1 flows from the holder cube of this second-level expanded reference cube through its port out into a pipe inside the upper plane of the first-level expansion which takes it to the addition predicate. The addition predicate receives the constant value 1 as a second argument, and returns the value 2, which flows out of its 'result' port and through a pipe to the port of the first-level expanded reference cube, and from there into the large holder cube. This constitutes the second solution to the query. It is easy to see how the expansion process can be continued, leading to an infinite number of solutions. Figure 20, contained in the next section, shows the evaluation of this query in progress. Note that all solutions are computed in parallel, so a single evaluation returns a set of solutions. The Cube user interface allows the user to then browse through the set of solutions (even as they are being computed), viewing them one at a time.

In summary, this example illustrates two key points. One of them is that a Cube query can have multiple solutions, just like a Prolog query. Cube, however, unlike Prolog, explores the paths leading to the various solutions in parallel, and is guaranteed to find every solution that can be found in finite time (i.e. by a finite number of 'expansions'). The second key aspect is that the logical notion of a resolution step—replacing a goal by the subgoals of a matching clause—has an intuitive visual counterpart, namely replacing a reference cube by the corresponding definition cube.

2.5. A Factorial Predicate

Figure 7 shows another example of a predicate definition cube. This cube defines the factorial predicate. Again, it consists of a transparent green cube, with an icon '!' on its top to name the predicate. Its two ports are set into the left and the right side of the outer cube, and are labeled 'n' and 'n!'. It contains two planes, the upper one representing the base case and the lower one the recursive case.

The upper plane contains two holder cubes. The left one is filled with the value 0

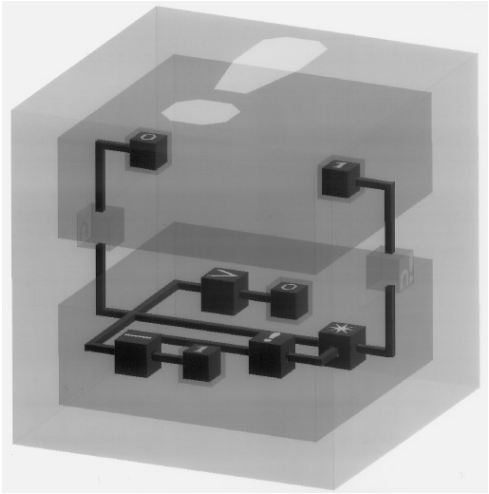

Figure 7. Definition of the factorial predicate

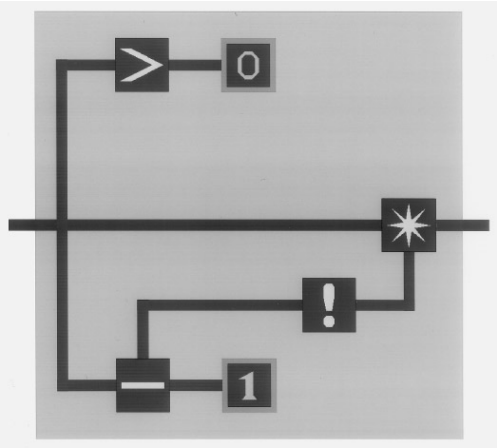

Figure 8. Close-up on the lower plane of Figure 7

and connected by a pipe to the left port ('n'); the right one is filled with the value 1 and connected to the right port ('n!').

The lower plane (shown by itself in Figure 8) contains a comparison predicate whose two arguments are connected by pipes to the left port ('n') and to a holder cube which contains the value 0. It also contains a subtraction predicate cube whose two 'input' arguments are connected to the left port ('n') and to a holder cube containing the value 1, and whose 'output' port is connected by a pipe to the 'input' port of a predicate cube which recursively refers to the factorial predicate. The 'output' port of the factorial predicate cube is connected to one of the 'input' ports of a multiplication predicate cube, whose other 'input' port is connected to the left port ('n'), while its 'output' port is connected to the right port of the definition cube ('n!').

Now consider the query shown in Figure 9, which contains a predicate cube referring to this definition, and where the user supplies a value, say v, to the left port of the definition cube ('n') of the factorial predicate. Again, we can imagine that the opaque reference cube gets replaced by ('expanded to') the transparent definition cube. The value v flows into the left port where it splits up, one copy of v flowing through a pipe into the upper plane and the other copy flowing through the other pipe into the lower plane.

The copy of v which goes to the upper plane flows into a holder cube which already contains the value 0. If v does not unify with 0, then the dataflow fails and with it the entire upper plane. One can imagine that it is simply taken out of the computation. Otherwise, the value 1 contained in the right holder cube flows out through a pipe and into the right port of the expanded reference cube (and possibly into an attached empty holder cube), thereby constituting a solution to the query.

The above description is intuitive but might mislead you into believing that the various dataflow operations are performed in some particular sequence, and that data flows into some prescribed direction. This is not the case. All flow of data takes place simultaneously and continues to happen until the system is in equilibrium (has reached a fixed-point). A nice analogy is pipes connecting containers with different

Figure 9. Program computing the factorial of 3

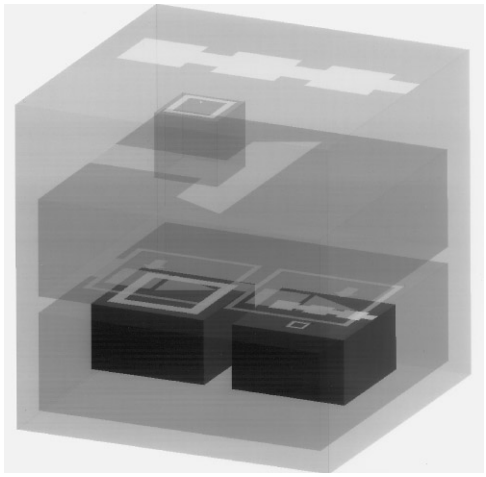

Figure 10. List type definition

air pressures (different amounts of information) inside. Once the pipes are opened (evaluation is started), air will flow through them until all connected containers have equal pressures. The same equilibrium is reached regardless of the sequence in which the pipes are opened, and the direction of air flow in a pipe is determined only by the pressure difference between two containers.

The second point that should be emphasized is that the result of a unification affects only the computation that happens, logically speaking, within the same conjunction. When the value v entered the port of the expanded reference cube, it was split up into two copies; one went to the upper, the other to the lower plane. A unification that refines v in the upper plane will not affect the copy of v in the lower plane.

Let us return to our example. As we said before, one copy of the value v flows through a pipe leading into the lower plane, where the pipe branches. One of its ends is connected to one port of a comparison predicate, whose other port receives the value 0. If v is not greater than 0, the comparison fails and with it the entire plane. The second end of the branching pipe is connected to a subtraction predicate, so v flows into the first argument of this predicate, which receives the value 1 as its second argument. The result of the subtraction, $v - 1$, flows out of the third argument port and into the port 'n' of a predicate cube recursively referring to the factorial predicate. The result of this computation, $(v - 1)!$, then flows out of the 'n!' port and into one of the two input ports of a multiplication predicate, whose other input port is connected to the third end of the branching pipe which carries v. The result of the multiplication, $v(v - 1)!$, finally flows out of the lower plane and into the port 'n!' of the expanded reference cube (and from there possibly into an attached empty holder cube).

2.6. Type Definitions

Most ML-like textual functional languages allow the programmer to define new types. The classical example is the definition of a polymorphic list type:

$$\mathsf{List}\ \alpha = \mathsf{nil} + \mathsf{cons}\ \alpha(\mathsf{List}\ \alpha)$$

defines two new constructors, nil and cons. A *constructor* is an uninterpreted function symbol. In this particular case, nil is a nullary function (i.e. a simple value). It is of type 'List of α', where α could be any type. cons is a binary function, which takes a value of type α as first and a value of type 'List of α' as second argument, and returns a new 'List of α'. Again, α ranges over all the types.

Note that the List type is defined in a recursive fashion: the nil constructor forms the base case, the cons constructor is the recursive case, as it creates a list by using another list. It is obvious that every finite list must be terminated by a nil constructor.

The expression nil denotes the empty list, that is, a list with no elements. The expression cons 1 nil denotes a list whose head is 1 and whose tail is the empty list; or, putting it differently, a list that has one element, namely 1. Similarly, the expression cons 1 (cons 2 nil) denotes a two-element list with 1 as the first and 2 as the second element.

List is referred to as a *type constructor,* an uninterpreted function that takes n types as arguments and returns a new type.

How can this notation be carried over to Cube? Constructors are functions, and as such they have arguments. In Cube, arguments (i.e. ports) are associated with icons. So, when defining a constructor, we not only need to specify the types of its arguments but also their icons. By symmetry, we also associate the arguments of type constructors with icons.

Figure 10 shows a *type definition cube* that defines the list type. It consists of a grey transparent cube with an icon on its top. The color grey distinguishes types from values, which are green. The icon names the type constructor that is to be defined (List in the textual definition).

The type definition cube has a *port* on its top side, which represents the one formal parameter of the type constructur (α in the textual definition). The port carries an icon on its outside which is used to distinguish it from other ports.

Inside the type definition cube are two grey transparent boxes, called *type planes,* that represent the two constructors of the list type. The planes are stacked on top of each other; in the context of types, vertical arrangement denotes a type sum, whereas horizontal arrangement denotes a type product. Each plane carries an icon on its top, identifying the constructor. The upper plane, which represents the nil constructor, is empty, as nil is a nullary constructor.

The lower plane represents the cons constructor; it contains two type cubes. The left type cube is a type reference cube that refers to the port of the enclosing definition cube, i.e. to α. Right above the cube, on the transparent wall of the enclosing box, is the icon for the first argument of cons.

The right type cube is a type constructor application. It consists of a type constructor cube that has a port and a type contained inside the port. The type constructor cube refers to the enclosing definition cube, i.e. List, and the type cube in its port refers to the port of the enclosing definition cube, i.e. α. So, the entire type constructor application denotes the type List α. Above the cube, on the transparent wall of the surrounding box, is the icon for the second argument of cons.

Note the similarity between type definition cubes and predicate definition cubes. Both are represented as transparent cubes with an icon on their top that names the object under definition. Ports on their side or top serve as formal parameters. Inside the definition cube are vertically stacked planes, which denote clauses for predicates

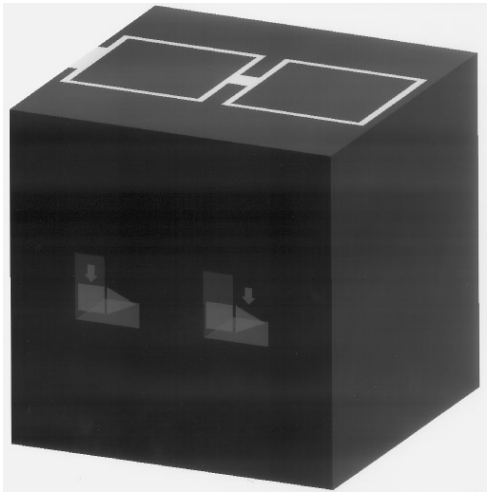

Figure 11. The '**cons**' constructor

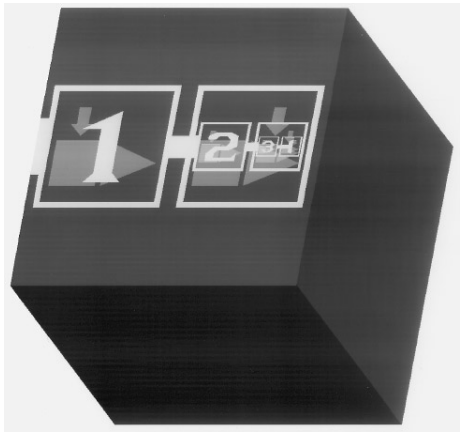

Figure 12. The list [1, 2, 3]

and variants for types. Vertical arrangement denotes '$\vee$' for predicates and '+' for types, while horizontal arrangement denotes '$\wedge$' for predicates and '$\times$' for types.

Type definition cubes can occur in a Cube program wherever predicate definition cubes may occur, and their scope extends just as far. Within this scope, there may be reference (i.e. term) cubes that refer to the constructors defined by the type definition cube. We call such reference cubes *constructor cubes.*

Figure 11 shows a constructor cube referring to the **cons** constructor defined by the list type definition cube from Figure 10. Like all value reference cubes, it is represented by an opaque green cube (colour not shown). It carries the **cons** icon on its top, the same icon that is on top of the lower plane of the type definition cube in Figure 10, and it has two ports in its side which are labeled by the same two icons as those which are hovering above the two type cubes in the lower plane. The left port (called the '**head**' port) can take an argument of any type, say τ, and the right port (called the '**tail**' port) can take an argument of type 'List of τ'.

Constructor cubes are first-class values, hence they can be contained in holder cubes, flow through pipes, be passed as arguments to predicates or to other constructors, etc. Their ports can be connected to pipes or be filled with values. Whenever we do the latter to build up complex structures, it is customary to move each port so that the icon which labels it occupies the same position as it has in the type plane defining the constructor. The result is a visually pleasing representation of recursive data structures. Figure 12 shows the list '[1, 2, 3]' enclosed in a holder cube. It shows that well-chosen constructor icons lead to a pleasing representation of recursive structures, and it explains why we chose the particular icons for **nil** and **cons**. We don't provide any special mechanisms for visualizing large structures, but rather rely on the user's ability to zoom in on a substructure.

Figure 13 shows how inferred list types are visualized. The program shows two holder cubes connected by a pipe. The left holder cube contains a cube representing the list '[1, 2, 3]'; the right holder cube is initially empty. Upon type inference, the

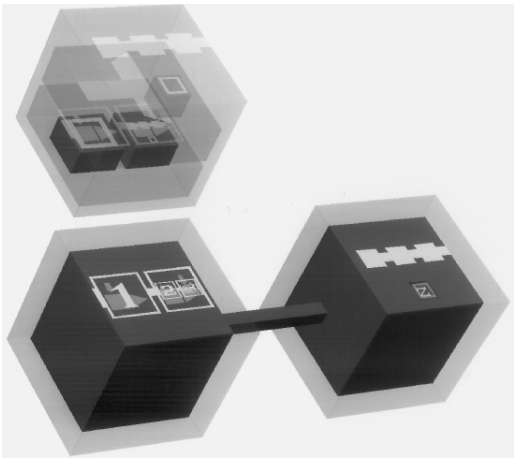

Figure 13. Interference of list types

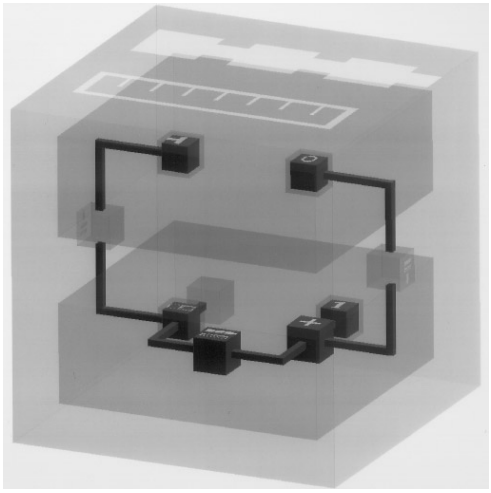

Figure 14. Predicate for computing the length of a list

Cube system infers that, since the left holder cube contains a list of integers and the two holder cubes are connected, the right holder cube is also restricted to lists of integers. Therefore it fills the right holder cube with a type cube representing lists of integers: a type reference cube referring to the list type constructor, whose port if filled with a type reference cube referring to the integer type constructor.

Constructors are used in Cube—just as in Prolog—both to construct terms and to deconstruct them. We have just seen that we can fill the ports of a constructor with term cubes to construct a larger term. Alternatively, we can connect them to pipes that supply them with values. The same technique is used for deconstruction, except that now values of flow *out of* the ports.

The remainder of this section shows two more predicate definitions and typical usages. In particular, we focus on first- and higher-order predicates over lists.

2.7. Determining the Length of a List

Consider the predicate definition cube shown in Figure 14, which takes two arguments, a list *l* and an integer *n,* and holds if the length of *l* is *n.* In the following, we will refer to the left port (which takes *l*) as the '**list**' port, and to the right port (which takes *n*) as the '**len**' port.

The upper plane contains the base case of the definition: the length of the empty list is `0`. This is represented by connecting the left port to a holder cube which contains the value **nil**, and the right port to a holder cube which contains the value `0`.

The lower plane (also shown in Figure 15) contains the recursive case: the length of a non-empty list is the length of its tail plus 1. This is expressed by connecting the left port to a holder cube which contains a **cons** constructor. **cons** is used here to deconstruct the list. Its '**head**' port is connected by a pipe to an empty holder cube, while its '**tail**' port is connected by a pipe to the '**list**' port of a recursive reference to the length predicate. The '**len**' port of the length predicate is connected by a pipe to one of the input ports of an addition predicate, whose other input port receives the

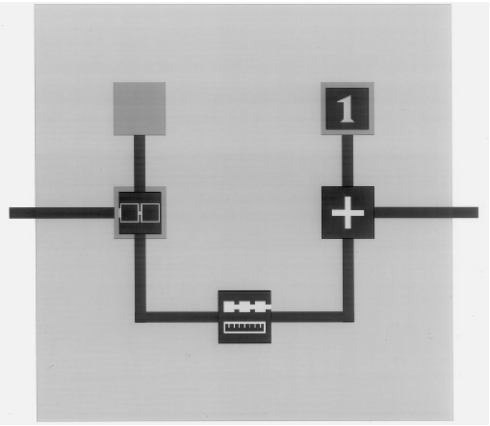

Figure 15. Close-up of the lower plane of Figure 14

value 1, and whose output port is connected by a pipe to the right port of the enclosing definition cube.

So, the cons constructor matches an incoming non-empty list, takes it apart, forwards its head to the empty holder cube (i.e. effectively discards it) and its tail to a recursive invocation of the length predicate, which thus determines the length of the tail of the incoming list. The addition predicate adds 1 to this length, yielding the length of the whole incoming list, and sends this value to the right port.

Figure 16 shows how this predicate can be used to compute the length of the list '[1, 2, 3]'. Upon evaluation, the empty right holder cube will be filled with the value 3.

Interestingly enough, the predicate can also be used with a reversed directionality. Instead of putting a list into the left holder cube and leaving the right one empty, we can put an integer, say 3, into the right holder cube and leave the left one empty. Figure 17 shows a closeup of the solution to this query: a list with 3 elements, each being a distinct uninstantiated variable, but all of them being of the same, however unknown, type.

2.8. Mapping a Predicate Over a List

This example shows the Cube definition of '*map*', a higher-order predicate that takes a binary predicate (say p) and two lists (say $[s_1, \ldots, s_m]$ and $[t_1, \ldots, t_n]$), and holds if both lists are of equal length (i.e. if $m = n$) and if the binary predicate holds when applied to corresponding elements in the two lists (i.e. if $p(s_i, t_i)$ holds for all $1 \leq i \leq m$).

Figure 18 shows the definition cube for this predicate. The port for the lower-order predicate is on the top of the predicate definition cube (which is the convention for predicate arguments), the ports for the two list arguments are on the left and the right.

The lower plane represents the base case: mapping any predicate over the empty list yields the empty list. This is expressed by connecting both 'list' ports to holder cubes which contain the value nil.

The upper plane represents the recursive case: both lists are decomposed, the lower-order predicate is applied to their heads, and map is applied recursively to their tails. This is expressed by two holder cubes, one being connected to the left port, the

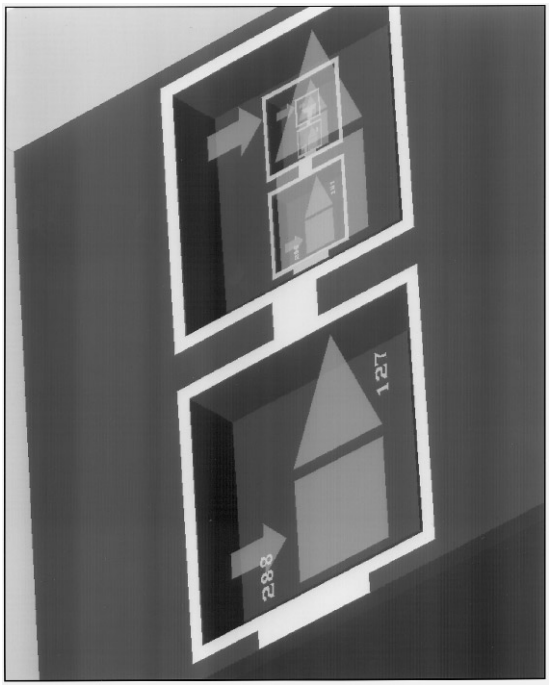

Figure 17. Computing a list of length 3 (close-up)

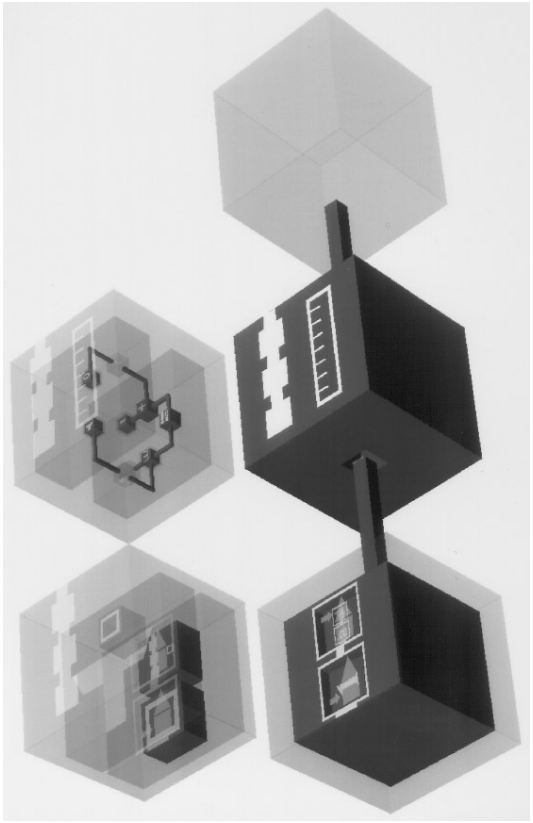

Figure 16. Computing the length of the list [1, 2, 3]

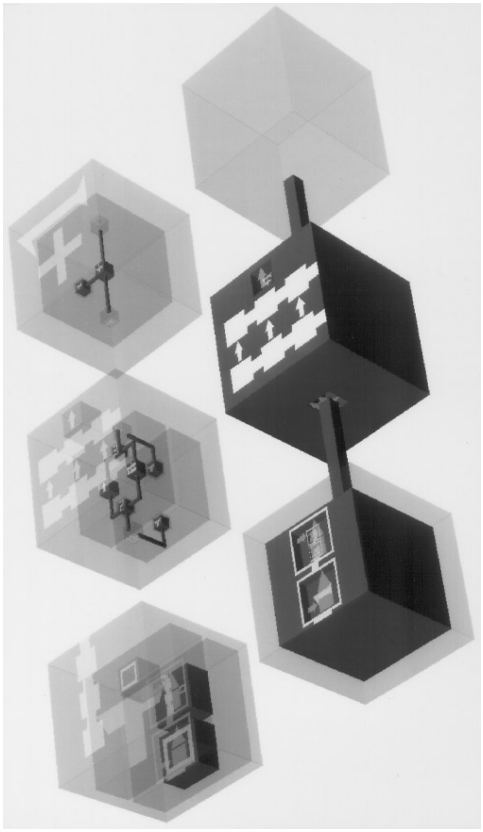

Figure 19. Mapping 'successor' over the list [1, 2, 3]

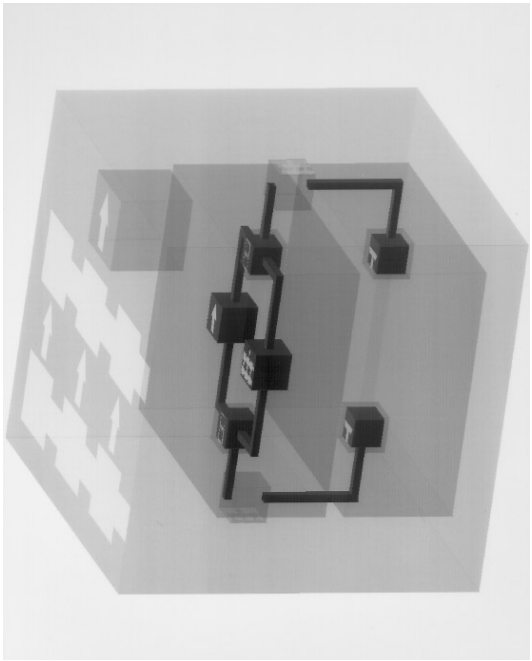

Figure 18. The 'map' predicate

other to the right port, and both being filled with a cons constructor. The 'head' and the 'tail' ports of both constructors are connected to pipes. The two pipes attached to the 'head' ports connect them to the two ports of a reference cube referring to the lower-order predicate; the two pipes attached to the 'tail' ports connect them to the two 'list' ports of a reference cube referring to map itself, the third port (the predicate argument) of this reference cube is filled with a reference to the lower-order predicate.

Figure 19 shows a query which uses the map predicate to map the successor predicate (represented by the definition cube at the top right) over the list '[1, 2, 3]'. Upon evaluation, the empty holder cube on the right will be filled with the list '[2, 3, 4]'.

The icons used inside the map predicate definition cube to identify the ports of the lower-order predicate are not the same as the icons identifying the two ports of the successor predicate. Hence, we need to 'relabel' the ports. This is done through a *port renaming cube*, a transparent cube (not to be confused with a holder cube) which surrounds a predicate cube or a constructor cube (map in this case) and carries the new icon on its transparent hull right above the port with the old icon.

The 'predicate' argument of map must always be completely ground; otherwise, the evaluation of the recursive case in which it is used will suspend until the predicate is ground. The allowable instantiation patterns of the two 'list' arguments, on the other hand, depends only on the predicate argument. If we use a predicate argument which expects both of its arguments to be gound (such as 'greater'), then both list arguments of map have to be completely ground; otherwise, the evaluation suspends. If we use a predicate argument which expects at least one of its arguments to be ground, then for each two corresponding elements of the two lists at least one has to be ground. Finally, if we use a predicate argument which expects neither of its arguments to be ground or even instantiated (such as 'equal'), then the two list arguments do not have to be instantiated at all. Instead, map will generate all possible solutions.

3. The Cube System

The first implementation of Cube, described in [24], consisted of a renderer, written in C, and an interpreter, written in Lazy ML. It served as a feasibility study and as a testbed for refining the syntax and semantics of Cube.

The second and current implementation, which was used to generate the figures in this paper, is written in Modula-3 [26], a modern procedural language that supports concurrency and provides garbage collection.

We used FormsVBT, the standard Modula-3 widget set, to build the 'traditional' parts of the user interface and wrote our own rendering engine for drawing 3D scenes. This renderer is written on top of X and does not require any 3D graphics hardware. When asked to redraw the scene, it performs a wireframe rendering of the scene. At the same time, it signals a background thread to generate a high-quality view of the scene. If the scene changes before the high-quality rendering is complete, the background thread is interrupted and restarted; otherwise, the wireframe rendering is replaced by the high-quality view. This approach ensures that the user can always interact with the system, without first having to wait for the high-quality rendering to finish.

Multi-threading is also used by the Cube interpreter to deal with infinite

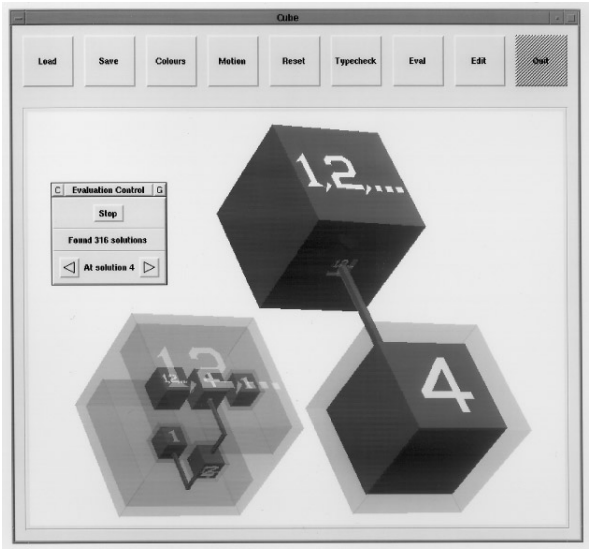

Figure 20. An evaluation in progress

computations. When the user presses the Eval button, a separate thread is created to perform the evaluation. At the same time, a control panel is popped up, which informs the user how many solutions have been found so far and allows him to interrupt the evaluation. In addition, this control panel allows him to browse through the various solutions, even while the evaluation is in progress. Figure 20 shows the program for generating all the natural numbers. The evaluation is in progress, the system has found 316 solutions so far and the user is currently looking at solution 4.

The current Cube system uses a mouse as the primary input device. This is somewhat of a limitation; a 2D pointing device such as a mouse can only be used to specify points in 2-space, which translates to a line rather than a point in 3-space. This means that either the user must perform two pointing operations to completely specify a point in 3-space, or that the system has to select a point on the line, possibly based on the current context and on existing objects in the scene.

Creating Cube programs and queries currently takes longer than it would take using a high-level textual language. However, much of the tediousness of program construction can be attributed to the fact that the system uses a 2D input device. A more appropriate input device, such as a dataglove, would speed up the editing process considerably.

4. A Review of the Language Design Choices

This section reviews some of the language design choices we have made. Cube was inspired by Kimura's Show-and-Tell [16], a functional language with a dataflow syntax. Show-and-Tell introduced a novel feature to the traditional dataflow model: inconsistency. Two conflicting values flowing into the same box of a Show-and-Tell dataflow diagram cause the box and its surroundings (delineated by a solid outline) to become *inconsistent.* Inconsistent areas are greyed out in the diagram and are taken

out of the computation. This is reminiscent of the notion of a *failed goal* in logic programming.

This observation prompted us to think about how Show-and-Tell would have to be modified to become a 'Visual Prolog'. In Prolog, a predicate definition consists of a set of clauses. Each clause can be expressed visually as a 2D dataflow diagram. Rather than using several disjoint 2D diagrams to express a predicate definition, we decided to integrate them into a coherent 3D diagram. This approach frees the programmer from having to mentally integrate several diagrams into a whole; thus, it moves workload from the cognitive to the perceptual level.

Stacking 2D dataflow diagrams on top of each other means that we use spatial layout to encode semantic content: the x and y dimensions span up a plane for 2D dataflow diagrams, the z dimension is used to stack multiple 2D dataflow diagrams. In a logic programming sense, predicates that lie in the same plane are conjoint (connected by AND), whereas stacked planes are disjoint (connected by OR).

We use the same visual metaphor for expressing type definitions. Type cubes that lie in the same xy plane are part of the same 'record' or 'tuple'; in type-theoretic terminology, they form a product type. Planes stacked in the z dimension are part of a variant record (akin to a union in C); in type-theoretic terminology, they form a sum type.

Research in type theory has uncovered a correspondence, known as the Curry–Howard isomorphism, between the logic operators AND and OR and the type operators PRODUCT and SUM. The similarity between the visual representations of predicate and type definitions matches the theoretic correspondence quite nicely.

The decision to use cubes (as opposed to, say, spheres or tetrahedra) as the basic shape was straightforward. Cubes are graphically simple: they are composed of six polygons, and can therefore be rendered much more efficiently than spheres. Furthermore, cubes are symmetric along three orthogonal axes. Tetrahedra (which are even easier to render) do not have this property. If we were to stack 2D diagrams inside a tedrahedron, the topmost diagram would have to be much smaller than the bottommost one.

We decided to restrict ourselves to two basic colors: grey for type cubes and green for value cubes. Our intention was to visually emphasize that numbers, constructors, and predicates are all first-class values. This decision is nice from a language purist's point of view; on the other hand, a richer coloring scheme that distinguished, say, between predicates and constructors, might actually be more comprehensible.

In textual languages, identifiers (e.g. procedure names or variables) provide the 'plumbing' for connecting operators and for relating definitions and uses of values. Cube offers two distinct mechanisms for this purpose: pipes and icons. Pipes (a defining feature of all dataflow languages) are used for connecting holder cubes and ports. One important advantage over textual variables is that pipes alleviate the need to come up with a new name or icon—they provide an implicit naming mechanism. Furthermore, given a pipe, one can see at a glance which operators it connects, while it is much harder to find all occurrences of a variable in a piece of textual code. This advantage is most pronounced for short pipes; excessively long pipes lead to the visual equivalent of 'spaghetti code' (pun intended). In Cube, individual predicate definitions are very compact so pipes are suitable to connect the objects inside a definition cube. However, predicate definitions can be far away from their uses. Therefore, we

provide an explicit naming mechanism—icons—to relate them. A predicate definition is represented by a transparent cube with an icon on its top; all opaque cubes in the same scope that carry this icon refer to the definition.

Enclosing diagrams with transparent cubes with superimposed icons makes programs visually more complex. However, much of the ambiguity can be resolved through motion parallax: interactively rotating the transparent cube helps the viewer in discriminating enclosing from enclosed objects. Moreover, a judicious choice of the transparency coefficient improves the recognition of both transparent and opaque structures, as a recent evaluation of transparent menus in 2D user interfaces suggests [12].

Cube does not make any provisions for animating evaluations. As it turns out, few if any animations would make sense. It is not sensible to animate dataflow by moving value cubes through pipes because, in the general case, data can flow in both directions at the same time (to use logic programming terminology, unifying two partially instantiated terms may refine both terms). Neither is it sensible to animate how results (such as lists) get built up during an evaluation, since the evaluation of each solution must be complete before we can be sure that it is indeed a solution.

In summary, we believe that the following aspects of our visual notation improve the comprehension of Cube programs:

- The use of the third dimension enables the integration of multiple 2D dataflow diagrams into a coherent whole, thereby shifting the programmer's mental workload from the cognitive to the perceptual level.
- Spatial layout is used to encode semantic information. Predicates that lie in the same *xy* plane are connected by a logical AND; planes that are stacked in the *z* dimension are connected by a logical OR.
- Color is used to distinguish types from values.
- Pipes are used to connect objects that are spatially close; icons are used to relate objects (such as predicate definitions and predicate uses) that are spatially farther apart.

Unfortunately, we did not verify the validity of these claims through any empirical study. This was largely due to time constraints: a prerequisite for any empirical study would be a much improved Cube editor. The existing editor is too primitive, both in terms of functionality and of convenience, to be usable by anyone but the author. A truly satisfactory editor would have to be freed from the constraints of a mouse-based user interface, and presumably be virtual-reality based—a project exceeding the scope of a PhD thesis.

5. Related Work

The design of Cube was quite heavily influenced by Show-and-Tell [16], a visual language based on the dataflow paradigm. In Show-and-Tell, constants, variables and operations are shown as *boxes.* Data flows from boxes to other boxes through *pipes,* which are depicted as arrows. A picture composed of boxes and pipes is called a *puzzle.* Show-and-Tell tries to *complete* this puzzle by performing every possible dataflow. If data flows into a box already containing a different value, the box becomes *inconsistent.* Inconsistency can be limited to a single box, or it can 'flow out'

of this box and turn its spatial environment inconsistent as well. Inconsistent areas are shaded grey and are considered to be removed from the diagram. If a pipe leads through an inconsistent area, no data can pass through it. This novel notion of consistency can be utilized in many ways, in particular, it fulfills the same purpose as a conditional or selection function in traditional textual languages. Cube generalizes the notions of completion and consistency to unification and satisfiability, respectively.

There are a few visual languages that have an explicit notion of types. Fabrik [20] and DataVis [13] use a type system that is similar to that of many procedural languages. They have a rich set of predefined types (records, arrays, etc), but do not allow the user to define new types. ESTL [22] was the first visual language to use the Hindley–Milner type system; variations of this system are used by VisaVis [28], an extension of Forms/3 [2], and Cube.

ESTL also introduced higher-order functions to visual programming. Other higher-order visual languages include DataVis [13], VPL [18], and *viz* [14].

There are a number of visual logic programming languages. The Transparent Prolog Machine [8] and a system by Senay and Lazzeri [33] both use And-Or-Trees to visualize Prolog programs and their execution. VPP [27] is a visual front-end for Prolog which uses labeled directed graphs. Pictorial Janus [15] is a visual notation for a concurrent constraint logic language. It uses diagram rewriting to animate the execution of programs, that is, it transforms a diagram into another diagram. VLP [17] is a visual logic programming language which represents clauses and literals as boxes, and uses spatial arrangement to denote conjunctions, disjunctions and implications. In this respect, it is very similar to Cube. However, it is not based on a dataflow notation but uses shared patterns to indicate shared variables. Finally, SPARCL [34] is founded on both logic and set theory.

Glinert suggested a decade ago that using a 3D syntax might increase the expressive power of visual languages [10]. To our best knowledge, Cube was the first visual language to use a 3D notation. By now, there are at least four more such languages: Lingua Graphica [37] provides a 3D syntax for C++ programs; CAEL-3D [30] provides a 3D syntax for a subset of Pascal; SPARCL [34], which is based on logic and set theory, recently adopted a 3D notation [35]; and MAP [9] uses spatial arrangements of nested cubes to represent data structures and execution order.

3D has also been used in a number of program visualization systems. Lieberman has built a system that uses 3D graphics to visualize the execution of Lisp programs [19]. The Plum system [31] uses 3D for laying out the nodes of a program's dynamic call graph. Recently, a number of algorithm animation systems have used 3D graphics, among them Polka-3D [36], Zeus3D [1], Pavane [6] and GASP [39].

6. Conclusion

This article described Cube, a three-dimensional visual programming language with a purely declarative, Horn-logic-based semantics. We gave an informal, example-driven overview of the language, and described the prototype implementation of the Cube interpreter and programming environment. Possible future research includes the design and construction of a virtual-reality based Cube programming environment.

For additional information on Cube, visit the Cube World-Wide Web home page at http://www.research.digital.com/SRC/personal/najork/cube.html.

Acknowledgments

Simon Kaplan and Eric Golin influenced all the stages of the design of Cube. Marc Brown and Allan Heydon provided helpful comments on various drafts of this paper. Thanks to all of them!

References

1. M. H. Brown & M. A. Najork (1993) Algorithm animation using 3D interactive graphics. In: *ACM Symposium on User Interface Software and Technology,* Altanta, GA. ACM Press, New York pp. 93–100.
2. M. Burnett (1993) Types and type inference in a visual programming language. In: *IEEE Symposium on Visual Languages,* Bergen, Norway. IEEE Computer Society Press, Los Alamitos, CA, pp. 238–243.
3. S.-K. Chang (ed.) (1990) *Visual Languages and Visual Programming* Plenum Press, New York. 340pp.
4. W. Chen, M. Kifer & D. S. Warren (1989) HiLog: a first-order semantics for higher-order logic programming constructs. In: *Logic Programming: Proceedings of the North American Conference 1989* (E. L. Lusk & R. A. Overbeck, eds) MIT Press, Cambridge, MA pp. 1090–1114.
5. W. F. Clocksin & C. F. Mellish (1991). *Programming in Prolog* Springer Verlag, Berlin.
6. K. C. Cox & G. C. Roman (1992) Abstraction in algorithm animation. In: *IEEE Workshop on Visual Languages,* Seattle, WA. IEEE Computers Society Press, Los Alamitos, CA, pp. 18–24.
7. L. Damas & R. Milner (1982) Principal type schemes for functional programs. In: *9th ACM Symposium on Principles of Programming Languages.* ACM Press, New York, pp. 207–212.
8. M. Eisenstadt & M. Brayshaw (1988) The transparent Prolog machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming* **5,** 227–342.
9. E. Freeman, D. Gelernter & S. Jagannathan (1995) In search of a simple visual vocabulary. In: *1995 IEEE Symposium on Visual Languages,* Darmstadt, Germany. IEEE Computer Society Press, Los Alamitos, CA, pp. 302–309.
10. E. P. Glinert (1987) Out of Flatland: towards 3-D visual programming. In: *1987 Fall Joint Computer Conference,* Dallas, TX. IEEE Computer Society Press, Los Alamitos, CA pp. 292–299.
11. E. P. Glinert (ed.) (1990) *Visual Programming Environments* (Volumes I and II) IEEE Computer Society Press, Los Alamitos, CA.
12. B. L. Harrison, G. Kurtenbach & K. J. Vincente (1995) An experimental evaluation of transparent user interface tools and information content. In: *ACM Symposium on User Interface Software and Technology,* Pittsburgh, PA. ACM Press, New York, pp. 81–90.
13. D. D. Hils (1991) DataVis: a visual programming language for scientific visualization. In: *1991 ACM Computer Science Conference,* San Antonio, TX. ACM Press, New York, pp. 439–448.
14. C. M. Holt (1990) *vis*: a visual language based on functions. In: *1990 IEEE Workshop on Visual Languages,* Skokie, IL. IEEE Computer Society Press, Los Alamitos, CA, pp. 221–226.
15. K. M. Kahn & V. A. Saraswat (1990) Complete visualizations of concurrent programs and their executions. In *IEEE Workshop on Visual Languages,* Skokie, IL. IEEE Computer Society Press, Los Alamitos, CA, pp. 7–15.
16. T. D. Kimura, J. W. Choi & J. M. Mack (1986) A visual language for keyboardless

programming. Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, MO.
17. D. Ladret & M. Rueher (1991) VLP: a visual programming language. *Journal of Visual Languages and Computing* **2,** 163–189.
18. D. Lau-Kee, A. Billyard, R. Faichney, Y. Kozata, P. Otto, M. Smith & I. Wilkinson (1991) VPL: an active declarative visual programming system. In: *IEEE Workshop on Visual Languages,* Kobe, Japan. IEEE Computer Society Press, Los Alamitos, CA, pp. 40–46.
19. H. Lieberman (1989) A three-dimensional representation for program execution. In: *IEEE Workshop on Visual Languages,* Rome, Italy. IEEE Computer Society Press, Los Alamitos, CA, pp. 111–116.
20. F. Ludolph, Y. Chow, D. Ingalls, S. Wallace & K. Doyle (1988) The Fabrik programming environment. In: *IEEE Workshop on Visual Languages,* Pittsburgh, PA. IEEE Computer Society Press, Los Alamitos, CA, pp. 222–230.
21. D. A. Miller & G. Nadathur (1986) Higher-order logic programming. In: *3rd International Conference on Logic Programming.* Published as *Lecture Notes in Computer Science* **225,** 448–462.
22. M. Najork & E. Golin (1990) Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. In: *IEEE Workshop on Visual Languages,* Skokie, IL. IEEE Computer Society Press, Los Alamitos, CA, pp. 215–220.
23. M. Najork & S. Kaplan (1991) The Cube language. In: *IEEE Workshop on Visual Languages,* Kobe, Japan. IEEE Computer Society Press, Los Alamitos, CA, pp. 218–224.
24. M. Najork & S. Kaplan (1992) A prototype implementation of the Cube language. In: *IEEE Workshop on Visual Languages,* Seattle, WA. IEEE Computer Society Press, Los Alamitos, CA, pp. 270–272.
25. M. Najork (1993) Programming in three dimensions. PhD Thesis. Technical Report UIUCDCS-R-93-1838, Department of Computer Science, University of Illinois at Urbana-Champaign.
26. G. Nelson (ed.) (1991) *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs.
27. L. F. Pau & H. Olason. Visual logic programming. *Journal of Visual Languages and Computing* **2,** 3–15.
28. J. Poswig, K. Teves, G. Vrankar & C. Moraga (1992) VisaVis—Contributions to theory and practice of highly interactive visual languages. In: *IEEE Workshop on Visual Languages,* Seattle, WA. IEEE Computer Society Press, Los Alamitos, CA, pp. 155–161.
29. G. Raeder (1985) A survey of current graphical programming techniques. *IEEE Computer* **18,** 11–25.
30. F. Van Reeth & E. Flerackers (1993) Three-dimensional graphical programming in CAEL. In: *IEEE Symposium on Visual Languages,* Bergen, Norway. IEEE Computer Society Press, Los Alamitos, CA, pp. 389–391.
31. S. P. Reiss (1995) An engine for the 3D visualization of program information. *Journal of Visual Languages and Computing* **6,** 229–323.
32. G. G. Robertson, S. K. Card & J. D. Mackinlay (1993) Information visualization using 3D interactive animation. *Communications of the ACM* **36,** 56–71.
33. H. Senay & S. G. Lazzeri (1991) Graphical representation of logic programs and their behavior. In: *IEEE Workshop on Visual Languages,* Kobe, Japan. IEEE Computer Society Press, Los Alamitos, CA, pp. 25–31.
34. L. Spratt & A. Ambler (1993) A visual logic programming language based on sets and partitioning constraints. In: *IEEE Symposium on Visual Languages,* Bergen, Norway. IEEE Computer Society Press, Los Alamitos, CA, pp. 204–208.
35. L. Spratt & A. Ambler (1994) Using 3D tubes to solve the intersecting line representation problem. In: *IEEE Symposium on Visual Languages,* St. Louis, MO. IEEE Computer Society Press, Los Alamitos, CA, pp. 254–261.
36. J. Stasko & J. Wehrli (1993) Three-dimensional computation visualization. In: *IEEE Symposium on Visual Languages,* Bergen, Norway. IEEE Computer Society Press, Los Alamitos, CA, pp. 100–107.

37. R. Stiles & M. Pontecorvo (1992) Lingua Graphica: a visual language for virtual environments. In: *IEEE Workshop on Visual Languages,* Seattle, WA. IEEE Computer Society Press, Los Alamitos, CA, pp. 225–227.
38. W. R. Sutherland (1966) On-Line Graphical Specification of Computer Procedures. Ph.D. Thesis, MIT, Cambridge, MA.
39. A. Tal & D. Dobkin (1995) Visualization of Geometric Algorithms. *IEEE Transactions on Visualization and Computer Graphics* **1,** 194–204.
40. C. Ware & G. Franck (1994) Viewing a graph in a virtual reality display is three times as good as a 2D diagram. In: *IEEE Symposium on Visual Languages,* St. Louis, MO. IEEE Computer Society Press, Los Alamitos, CA, pp. 182–183.